



The Process Specification Language (PSL) Overview and Version 1.0 Specification

Craig Schlenoff**Josh Lubell**

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

Michael Gruninger

4 Taddle Creek Road
University of Toronto
Toronto, Ontario M5S 3G9

Florence Tissot

Knowledge Based Systems, Inc.
1408 University Drive East
College Station, TX 77840

John Valois

STEPTools Inc.
Rensselaer Technology Park
Troy, NY 12180

Jintae Lee

University of Colorado
Campus Box 419
Boulder, Colorado 80309

NIST

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and
Technology

QC
100

.U56

NO.6459

2000

The Process Specification Language (PSL) Overview and Version 1.0 Specification

Craig Schlenoff

Josh Lubell

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

Michael Gruninger

4 Taddle Creek Road
University of Toronto
Toronto, Ontario M5S 3G9

Florence Tissot

Knowledge Based Systems, Inc.
1408 University Drive East
College Station, TX 77840

John Valois

STEPTools Inc.
Rensselaer Technology Park
Troy, NY 12180

Jintae Lee

University of Colorado
Campus Box 419
Boulder, Colorado 80309

February 2000



U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Dr. Cheryl L. Shavers, Under Secretary
of Commerce for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

The Process Specification Language (PSL) Overview and Version 1.0 Specification

Craig Schlenoff
National Institute of Standards and Technology
100 Bureau Drive - Stop 8260
Gaithersburg, MD 20899

Michael Gruninger
4 Taddle Creek Road
University of Toronto
Toronto, Ontario M5S 3G9

Florence Tissot
Knowledge Based Systems, Inc.
1408 University Drive East
College Station, TX 77840

John Valois
STEPTools Inc.
Rensselaer Technology Park
Troy, NY 12180

Josh Lubell
National Institute of Standards and Technology
100 Bureau Drive - Stop 8260
Gaithersburg, MD 20899

Jintae Lee
University of Colorado - Campus Box 419
Boulder, Colorado 80309

Keywords: manufacturing process specification language, PSL, KIF, ontology

This document describes Version 1.0 of the Process Specification Language (PSL). PSL is an interchange format designed to help exchange process information automatically among a wide variety of manufacturing applications such as process modeling, process planning, scheduling, simulation, workflow, project management, and business process re-engineering tools. These tools would interoperate by translating between their native format and PSL. Then, any system would be able to automatically exchange process information with any other system via PSL.

This document focuses specifically on PSL's rationale, semantic architecture, informal documentation, and the vision of how one would translate in and out of PSL.

1	Overview	4
1.1	Purpose	4
1.2	Approach	4
1.3	Scope	5
2	Related Work	5
3	The Process Specification Language	7
3.1	The Need for Semantics	7
3.2	What is PSL?	8
3.2.1	The Language	9
3.2.2	Model Theory	9
3.2.3	Proof Theory	10
3.3	Informal Semantics of PSL Core	12
3.4	Extensions in PSL 1.0	13
3.4.1	PSL Outer Core	13
3.4.2	Generic Activities and Ordering Relations	14
3.4.3	PSL Extensions for Schedules	14
3.5	Approach for Developing Extensions	15
4	Informal Documentation	15
4.1	Introduction	15
4.2	PSL Core	16
4.2.1	Kinds for the PSL Core	16
4.2.2	Individuals for the PSL Core	18
4.2.3	Primitive Relations for the PSL Core	19
4.2.4	Primitive Functions for the PSL Core	19
4.2.5	Defined Relations for the PSL Core	20
4.2.6	Definitions and Axioms for the PSL Core in the formal language	21
4.2.7	PSL Core Axioms	22
4.3	Subactivity Extension	24
4.3.1	Defined Classes in the Subactivity Extension	24
4.3.2	Defined Relations in the Subactivity Extension	25
4.3.3	Formal Axioms in the Subactivity Extension	25
4.4	Activity-Occurrences Extension	26
4.4.1	Introduced Relations in the Activity-Occurrences Extension	26
4.4.2	Defined Relations in the Activity-Occurrences Extension	27
4.4.3	Formal Axioms in the Activity-Occurrences Extension	27
4.5	States Extension	29
4.5.1	Classes of Objects in the States Extension	29
4.5.2	Introduced Relations in the States Extension	29
4.6	Integer and Duration Extension	29
4.6.1	Primitive Kinds in the Integer Extension	30
4.6.2	Defined Kinds in the Integer Extension	30
4.6.3	Individuals in the Integer Extension	30
4.6.4	Functions in the Integer Extension	30
4.6.5	Relations on Integers	31
4.6.6	Formal Definitions and Axioms for Integers	32
4.6.7	Primitive Kinds in the Timedurations Extension	35
4.6.8	Individuals in the Timeduration Extension	35
4.6.9	Defined Properties and Relations in the Duration Extension	35
4.6.10	Defined Functions in the Duration Extension	36
4.6.11	Functions in the Duration Extension	36

4.7	Ordering Relations over Activities Extension	40
4.7.1	Classes of Activities in the Ordering Relations Extension	40
4.7.2	Relations in the Ordering Relations Extension	41
4.8	Ordering Relations For Complex Sequences of Activities Extension	41
4.8.1	Defined Relations in the Ordering Relations For Complex Sequences Extension	41
4.9	Nondeterministic Activities Extension	43
4.9.1	Classes of Activities in the Nondeterministic Activities Extension	43
4.9.2	Formal Axioms in the Nondeterministic Activities Extension	44
4.10	Reasoning about States Extension	44
4.10.1	Classes of Fluents in the Reasoning about States Extension	44
4.10.2	Relations In the Reasoning about States Extension	45
4.10.3	Formal Definitions for the Reasoning about States Extension	46
4.11	Interval Activities Extension	49
4.11.1	Classes of Activities in the Interval Activities Extension	49
4.11.2	Defined Functions in the Interval Activities Extension	50
4.11.3	Defined Relations in the Interval Activities Extension	50
4.11.4	Formal Definitions and Axioms for the Interval Activities Extension	50
4.12	Temporal Ordering Relations Extension	52
4.12.1	Defined Relations in the Temporal Ordering Extension	52
4.12.2	Formal Definitions for the Temporal Ordering Extension	53
4.13	Junctions Extension	54
4.13.1	Classes of Activities in the Junctions Extension	54
4.13.2	Formal Definitions for the Junctions Extension	55
5	Translation Using PSL	57
5.1	Motivation	57
5.2	Overview of Semantic and Syntactic Translation	57
6	Conclusion	60
7	References	62
	Appendix A: Sample PSL Instance	64
	Appendix B: Mapping PSL Concepts to the EXPRESS Representation	72
	Mapping the PSL Ontology to EXPRESS	72
	Use of EXPRESS-X	74
	Appendix C: Mapping PSL Concepts to the eXtensible Markup Language (XML)	76
	Representation	76
	XML's Strengths and Weaknesses as a Presentation Language for PSL	76
	Guidelines for Mapping PSL to XML	76
	Appendix D: Basic PSL Syntax	80
	BNF Conventions	80
	Basic Tokens and Syntactic Categories	80
	Lexicons	81
	Grammars	81
	Languages	82
	Defined Quantifiers	82

1 Overview

1.1 Purpose

As the use of information technology in manufacturing operations has matured, the capability of software applications to interoperate has become increasingly important. Initially, translation programs were written to enable communication from one specific application to another, although not necessarily both ways. As the number of applications has increased and the information has become more complex, it has become much more difficult for software developers to provide translators between every pair of applications that need to exchange information. Standards-based translation mechanisms have simplified integration for some manufacturing software developers by requiring only a single translator to be developed between their respective software product and the interchange standard. By developing only this single translator, the application can interoperate with a wide variety of other applications that have a similar translator between that standard and their application.

This challenge of interoperability is especially apparent with respect to manufacturing process information. Many manufacturing engineering and business software applications use process information, including manufacturing simulation, production scheduling, manufacturing process planning, workflow, business process reengineering, product realization process modeling, and project management. Each of these applications utilizes process information in a different way, so it is not surprising that these applications' representations of process information are different as well. The primary difficulty with developing a standard to exchange process information is that these applications sometimes associate different meanings with the terms representing the information that they are exchanging. For example, in the case of a workflow system, a resource is primarily thought of as the information that is used to make necessary decisions. In a process planning system, a resource is primarily thought of as a person or machine that will perform a given task. If one were to integrate a process model from a workflow with a process planning application, one's first inclination would most likely be to map one resource concept to the other. This mapping would undoubtedly cause confusion. Therefore, both the semantics and the syntax of these applications need to be considered when translating to a neutral standard. In this case, the standard must be able to capture all of the potential meanings behind the information being exchanged.

The Process Specification Language (PSL) project at the National Institute of Standards and Technology (NIST) is addressing this issue by creating a neutral, standard language for process specification to serve as an Interlingua to integrate multiple process-related applications throughout the manufacturing life cycle. This interchange language is unique due to the formal semantic definitions (the ontology) that underlie the language. Because of these explicit and unambiguous definitions, information exchange can be achieved without relying on hidden assumptions or subjective mappings.

1.2 Approach

The approach in developing the PSL involved five phases: requirements gathering, existing process representation analysis, language creation, pilot implementation and validation, and submission as a candidate standard. The completion of the first phase

resulted in a comprehensive set of requirements for specifying manufacturing processes [1]. In the second phase, twenty-six process representations were identified as candidates for analysis by the PSL team and analyzed with respect to the phase one requirements [2]. Nearly all of the representations studied focused on the syntax of process specification rather than the meaning of terms, the semantics. While this is sufficient for exchanging information between applications of the same type, such as process planning, different types of applications associate different meanings with similar or identical terms. As a result of this, a large focus of the third phase involved the development of a formal semantic layer (an ontology) for PSL based on the Knowledge Interchange Format (KIF) specification [3]. By using this ontology to define explicitly and clearly the concepts intrinsic to manufacturing process information, PSL was used to integrate multiple existing manufacturing process applications in the fourth phase of the project.

1.3 Scope

To keep this work feasible, the scope of study is limited to the realm of discrete processes related to manufacturing, including all processes in the design/manufacturing life cycle. Business processes and manufacturing engineering processes are included in this work both to ascertain common aspects for process specification and to acknowledge the current and future integration of business and engineering functions.

In addition, the goal of this project is to create a “*process specification language*,” not a “*process characterization language*.” Our definition of a *process specification language* is a language used to specify a process or a flow of processes, including supporting parameters and settings. This may be done for prescriptive or descriptive purposes and is composed of an ontology and one or more presentations. This is different from a *process characterization language*, which we define as a language describing the behaviors and capabilities of a process independent of any specific application. For example, the dynamic or kinematic properties of a process (e.g., tool chatter, a numerical model capturing the dynamic behavior of a process or limits on the process’s performance or applicability), independent of a specific process, would be included in a characterization language.

2 Related Work

PSL is a neutral language for process specification to serve as an interchange language to integrate multiple process-related applications throughout the manufacturing process life cycle (from initial process conception all the way through to process retirement). This project is related to, and in many cases working closely with, many other efforts. These include individual efforts (those involving only a single company or academic institution) such as A Language for Process Specification (ALPS) Project [4], the Toronto Virtual Enterprise (TOVE) Project [5], the Enterprise Ontology Project [6], and the Core Plan Representation (CPR) Project [7]. In addition, the PSL project is in close collaboration with various projects (those that involve numerous companies or academic institutions) such as Shared Planning and Activity Representation (SPAR) Project [8], the Process Interchange Format (PIF) Project [9], and the WorkFlow Management Coalition (WfMC) [10].

ALPS was a NIST research project whose goal was to identify information models to facilitate process specification and to transfer this information to process control. The

PSL project, which could be viewed as a spin-off of the ALPS project, has a goal to take a much deeper look into the issues of process specification and to explore these issues in a much broader set of manufacturing domains.

The TOVE project provides a generic, reusable data model that provides a shared terminology for the enterprise that each agent can jointly understand and use. The Enterprise Ontology project's goal is to provide "a collection of terms and definitions relevant to business enterprises to enable coping with a fast changing environment through improved business planning, greater flexibility, more effective communication and integration." While both TOVE and the Enterprise Ontology focus on business processes, there are common semantic concepts in both these projects and the manufacturing process-focused PSL.

The CPR project is attempting to develop a model that supports the representation needs of many different military-planning systems. The SPAR project is an ARPI- (ARPA (Advanced Research Projects Agency)/Rome Laboratory Planning Initiative) funded project whose goals are similar to CPR. Both of these projects are similar to PSL in the sense that they are attempting to create a shared model of what constitutes a plan, process, or activity. There has even been coordination between the participants in SPAR, CPR, and PSL. The core models have similar roots. However, both SPAR and CPR are focusing more on military types of plans and processes.

PIF is an interchange format based upon formally defined semantic concepts, like PSL. However, unlike PSL, PIF is focused on modeling business processes and offers a single, syntactical presentation, the BNF (Backus-Naur Format) specification of the Ontolingua¹ Frame syntax.

The Workflow Management Coalition has developed a Workflow Reference Model whose purpose is to identify the characteristics, terminology, and components to enable the development and interoperability of workflow specifications. Although the area of workflow is within the scope of the PSL project, it is only one small component. The Workflow Reference Model has and will be used by the PSL project to ensure consistency.

In addition to the existing projects described above, there have been countless, previous efforts to create process representations focusing specifically on various representational areas or on different functionality. For example, representational areas such as workflow, process planning, artificial intelligence planning, and business process re-engineering have had representations developed focusing solely on their respective areas. Equally important to the representational area in which the representations are being developed is the role (functionality) that the representation will play. There have been process representations developed which have focused merely on graphically documenting a process, to those which are used as internal representations for software packages, to those which are used as a neutral representation to enable integration. The process representations that resulted from many of these efforts were analyzed in the second phase of the PSL project (described above). A sampling of some of these existing

¹ No approval or endorsement of any commercial product in this paper by the National Institute of Standards and Technology is intended or implied. This paper was prepared by United States Government employees as part of their official duties and is, therefore, a work of the U.S. Government and not subject to copyright.

process representations is shown in Figure 1. For more information about the representations listed in the figure, please see [2].

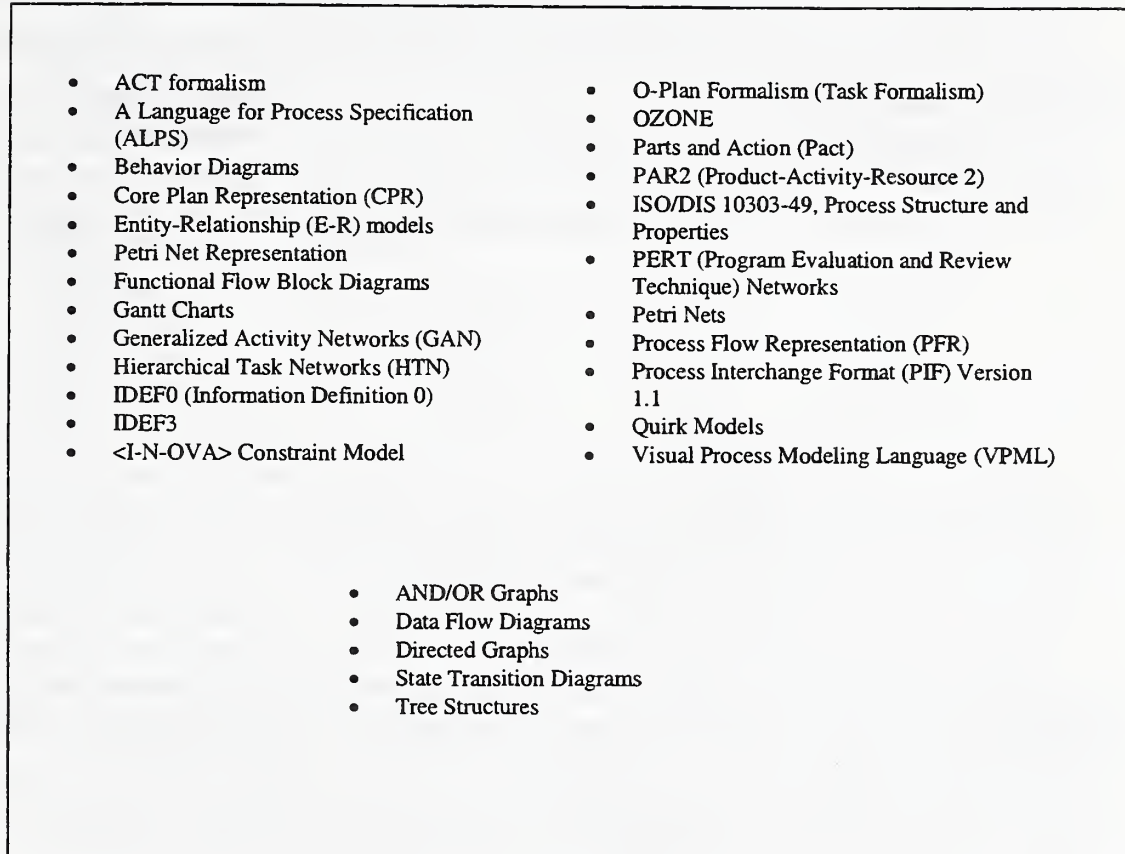


Figure 1: A Sampling of Existing Process Representations

3 The Process Specification Language

3.1 The Need for Semantics

Existing approaches to process modeling lack an adequate specification of the semantics of the process terminology, which leads to inconsistent interpretations and uses of information. Analysis is hindered because models tend to be unique to their applications and are rarely reused. Obstacles to interoperability arise from the fact that the systems that support the functions in many enterprises were created independently, and do not share the same semantics for the terminology of their process models.

For example, consider Figure 2 in which two existing process planning applications are attempting to exchange data. Intuitively the applications can share concepts; for example, both *material* in Application A and *workpiece* in Application B correspond to a common concept of *work-in-progress*. However, without explicit definitions for the terms, it is difficult to see how concepts in each application correspond to each other. Both Application A and B have the term *resource*, but in each application this term has a different meaning. Simply sharing terminology is insufficient to support interoperability;

the applications must share their semantics, i.e., the *meanings* of their respective terminologies.

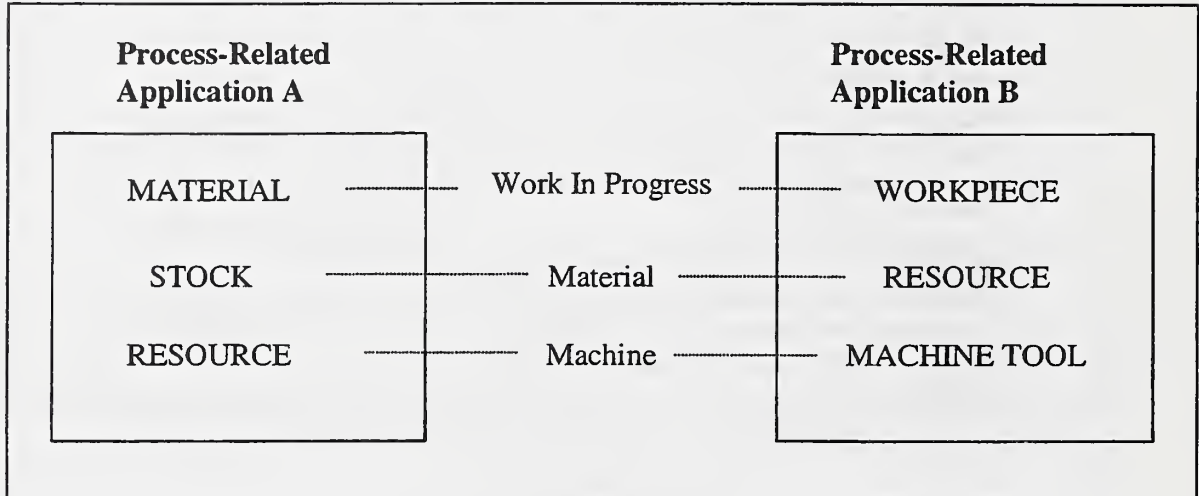


Figure 2: Why Semantics?

A rigorous foundation for process design, analysis, and execution therefore requires a formal specification of the semantics of process models. One approach to generating this specification is through the use of *ontologies*. An ontology is a formal description of the entities within a given domain: the properties they possess, the relationships they participate in, the constraints they are subject to, and the patterns of behavior they exhibit [11]. It provides a common terminology that helps to capture key distinctions among concepts in different domains, which aids in the translation process.

A goal of PSL is to facilitate application interoperability by means of the development of translators between native formats of those applications and PSL. Without an overarching language like PSL to serve as a medium of information interchange between applications, a unique translator must be written for every two-party exchange. However, this approach requires $n(n-1)$ translators for n different ontologies. With PSL serving as a standardized medium of information interchange, the number of translators for n different ontologies is reduced to n , since it only requires translators between native ontologies and the interchange ontology. The other feature of this approach is that the applications interact primarily through the exchange of files that contain process information. This requires the explicit specification of the semantics of these process descriptions. There can be no procedural interpretation of the application constructs or any implicit assumptions about process behavior. Similarly, all assumptions made by the application must be made explicit since translation must be done using the input file alone.

3.2 What is PSL?

An *ontology* is a set of specialized terminology along with some specification of the meaning of terms in the lexicon. The primary component of PSL is an ontology designed

to represent the primitive concepts that, according to PSL, are adequate for describing basic manufacturing, engineering, and business processes. Note that the focus of an ontology is not only on terms, but also on their meaning. We can include an arbitrary set of terms in our ontology, but they can only be shared if we agree on their meaning. It is the intended *semantics* of the terms that is being shared, *not simply* the terms.

The challenge is that a framework is needed to make the meaning of the terminology for ontologies explicit. Any intuitions that are implicit are a possible source of ambiguity and confusion. For the PSL ontology, we must provide a rigorous mathematical characterization of process information as well as precise expression of the basic logical properties of that information in the PSL language. In providing the ontology, we therefore specify three notions:

- language
- model theory
- proof theory

3.2.1 The Language

A language is a lexicon (a set of symbols) and a grammar (a specification of how these symbols can be combined to make well-formed formulas). The lexicon consists of logical symbols (such as boolean connectives and quantifiers) and nonlogical symbols. For PSL, the nonlogical part of the lexicon consists of expressions (constants, function symbols, and predicates) chosen to represent the basic concepts in the PSL ontology. Notably, these will include the 1-place predicates 'activity', 'activity-occurrence', 'object', and 'timepoint' for the four primary kinds of entity in the basic PSL ontology, the function symbols *beginof* and *endof* that return the timepoints at which an activity begins and ends, respectively, and the 2-place predicates *is-occurring-at*, *occurrence-of*, *exists-at*, *before*, and *participates-in*, which express important relations between various elements of the ontology.

The underlying grammar used for PSL is based roughly on the grammar of KIF (Knowledge Interchange Format). KIF is a formal language based on first-order logic developed for the exchange of knowledge among different computer programs with disparate representations. KIF provides the level of rigor necessary to define concepts in the ontology unambiguously, a necessary characteristic to exchange manufacturing process information using the PSL Ontology. Like KIF, PSL provides a rigorous BNF (Backus-Naur form) specification. The BNF provides a rigorous and precise recursive definition of the class of grammatically correct expressions of the PSL language. In addition to the simple clarity of such a definition, the BNF definition makes it possible to develop computational tools for the transfer of process information, one of PSL's central goals. In particular, by fixing the definition of the language precisely (and *only* by so fixing its definition), it is possible to develop translators between PSL and other, similarly well-defined representation languages. The actual PSL BNF can be found below in Appendix D.

3.2.2 Model Theory

The model theory of PSL provides a rigorous, abstract mathematical characterization of the semantics, or meaning, of the language of PSL. This representation is typically a set with some additional structure (e.g., a partial ordering, lattice, or vector space). The

model theory then defines meanings for the terminology and a notion of truth for sentences of the language in terms of this model. The objective is to identify each concept in the language with an element of some mathematical structure, such as lattices, linear orderings, and vector spaces.

Given a model theory, the underlying theory of the mathematical structures used in the theory then becomes available as a basis for reasoning about the concepts intended by the terms of the PSL language and their logical relationships, so that the set of models constitutes the formal semantics of the ontology.

3.2.3 Proof Theory

The proof theory consists of three components: PSL Core, one or more foundational theories, and PSL extensions.

PSL Core

The PSL Core is a set of axioms written in the basic language of PSL. The PSL Core axioms provide a syntactic representation of the PSL model theory, in that they are sound and complete with regard to the model theory. That is to say, every axiom is true in every model of the language of the theory, and every sentence of the language of PSL that is true in every model of PSL can be derived from the axioms. Because of this tight connection between the Core axioms and the model theory for PSL, the Core itself can be said to provide a *semantics* for the terms in the PSL language. (And indeed, in this document, we will frequently speak of a set of axioms “providing semantics” for a given lexicon.)

Foundational Theories

The purpose of PSL Core is to axiomatize a set of intuitive semantic primitives that is adequate for describing basic processes. Consequently, its characterization of them does not make many assumptions about their nature beyond what is needed for describing those processes. The advantage of this is that the account of processes implicit in PSL Core is relatively straightforward and uncontroversial. However, a corresponding liability is that the Core is rather weak in terms of pure logical strength. In particular, the theory is not strong enough to provide definitions of the many auxiliary notions that become needed to describe an increasingly broader range of processes in increasingly finer detail. (Auxiliary notions are axiomatized in PSL *extensions*, discussed next.) For this reason, PSL includes one or more *foundational theories*. A foundational theory is a theory whose expressive power is sufficient for giving precise definitions of, or axiomatizations for, the primitive concepts of PSL, thus greatly enhancing the precision of semantic translations between different schemes. Moreover, in a foundational theory, one can define a substantial number of auxiliary terms, and prove important metatheoretical properties of the core and its extensions.

There are several good foundational theories. Of these, set theory is perhaps the most familiar, and perhaps, all in all, the most powerful. Set theory’s foundational capabilities are well known. It is, in particular, capable of serving as a foundation for all of classical mathematics, in the sense that all notions of classical mathematics – integers, real

numbers, topological spaces, etc. – can be defined as sets of a certain sort and, under those definitions, their classical properties derived as theorems of set theory.

For PSL's purposes, however, a more suitable foundation is a modified and extended variation of the *situation calculus*. The reason for this is that the situation calculus's own primitives – *situation*, *action*, *fluent* (roughly, *proposition*) – are already highly compatible with the primitives of PSL; indeed, it is very natural to identify PSL primitives with, or define them in terms of, the primitives of the situation calculus. In addition, the situation calculus is also strong enough to define a wide variety of auxiliary notions and, with the addition of some set theory, it can be used as a basis for proving basic metatheoretic results about the Core and its extensions as well.

Extensions

The third component of PSL are the *extensions*. A PSL extension gives one the resources to express information involving concepts that are not part of PSL Core. Extensions give PSL a clean, modular character. PSL Core is a relatively simple theory that is adequate for expressing a wide range of basic processes. However, more complex processes require expressive resources that exceed those of PSL Core. Rather than clutter PSL Core itself with every conceivable concept that might prove useful in describing one process or another, a variety of separate, modular extensions have been (and continue to be) developed that can be added to PSL Core as needed. In this way a user can tailor PSL precisely to suit his or her expressive needs.

To define an extension, new constants and/or predicates are added to the basic PSL language, and, for each new linguistic item, one or more axioms are given that constrain its interpretation. In this way one provides a “semantics” for the new linguistic items. A good example of such an extension is the theory of timedurations below. PSL Core itself does not provide the resources to express information about timedurations. However, in many contexts, such a notion might be useful or even essential. Consequently, a theory of timedurations has been developed which can be added as to PSL Core, thus providing the user with the desired expressive power.

When combined with a foundational theory like the situation calculus, a distinction can be drawn between *definitional* and *nondefinitional* extensions. As the name suggests, a definitional extension is an extension whose new linguistic items can be completely defined in terms of the foundational theory and PSL Core. Theoretically, then, definitional extensions add no new expressive power to PSL Core + foundational theory, and hence involve no new theoretical overhead. However, because definitions of many subtle notions can be quite involved, definitional extensions can prove extremely useful for describing complex processes in as succinct a manner as possible. Nondefinitional extensions, of course, are extensions that involve at least one notion that cannot be defined in terms of PSL Core and the chosen foundational theory.

The three components of the PSL architecture and their relations are illustrated in Figure 3. The solid arrows indicate the definability relation. The dashed lines indicate partial definability, i.e., the case where some, but not all the additional linguistic items in the language of an extension are definable. Two or more solid arrows pointing to the same

oval indicate the possibility that more than one given theory might jointly be used to define a new extension. Therefore, we might have connected PSL Core to foundational theories, but this would not sufficiently distinguish the central role of the Core from the more auxiliary roles of extensions. Hence, we picture PSL Core as sitting directly upon the foundational theories. “(+ Foundational Theory)” in the PSL Core box indicates that PSL Core together with a foundational theory are typically used to formulate definitional extensions.

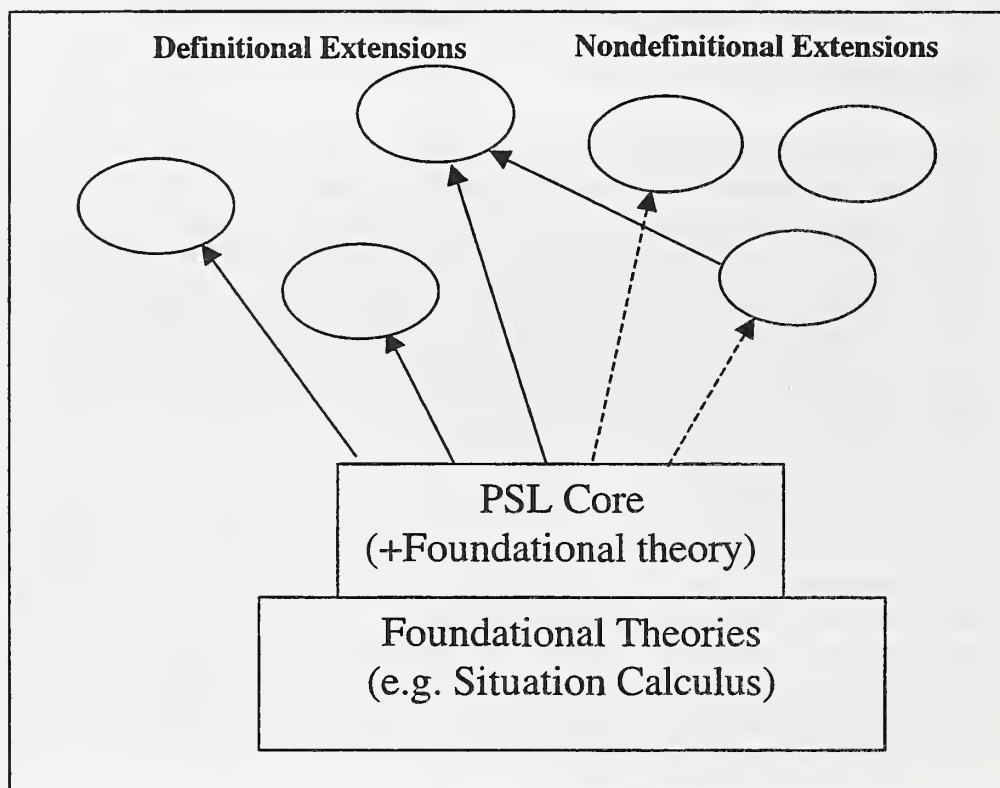


Figure 3: The PSL Semantic Architecture

3.3 Informal Semantics of PSL Core

PSL Core is based upon a precise, mathematical, first-order theory, i.e., a formal language, a precise mathematical semantics for the language, and a set of axioms that express the semantics in the language. Here we will provide a brief informal sketch of the semantics. There are four primitive classes, two primitive functions, and three primitive relations in the ontology of PSL Core. The classes are OBJECT, ACTIVITY, ACTIVITY_OCCURRENCE and TIMEPOINT. The four relations are PARTICIPATES-IN, BEFORE, and OCCURRENCE-OF. The two functions are BEGINOF, and ENDOF. ACTIVITIES, ACTIVITY_OCCURRENCES, TIMEPOINTS (or "POINT"s, for short), and OBJECTs are known collectively as entities, or things. These classes are all pairwise disjoint.

Intuitively, an OBJECT is a concrete or abstract thing that can participate in an ACTIVITY. The most typical examples of OBJECTs are ordinary, tangible things, such

as people, chairs, car bodies, NC-machines, though abstract objects, such as numbers, are not excluded. OBJECTs can come into existence (e.g., be created) and go out of existence (e.g., be “used up” as a resource) at certain points in time. In such cases, an OBJECT has a begin and/or end point. Some OBJECTs, e.g., numbers, do not have finite begin and end points. In some contexts it may be useful to model certain ordinary OBJECTs as having no such points either.

An ACTIVITY-OCCURRENCE is a limited, temporally extended piece of the world, such as the first mountain stage of the 1997 Tour de France or the eruption of Mt. St. Helen. Any ACTIVITY-OCCURRENCE is simply taken to be characterized chiefly by two things: its temporal extent, as determined by its begin and end POINTs (possibly at infinity), and the set of OBJECTs that participate in that ACTIVITY at some point between its begin and end POINTs.

TIMEPOINTS are ordered by the BEFORE relation. This relation is transitive, non-reflexive, total ordering. In PSL Core, that time is not dense (i.e., between any two distinct TIMEPOINTS there is a third TIMEPOINT), though it is assumed that time is infinite. POINTs at infinity (INF+ and INF-) are assumed for convenience. (Denseness, of course, could easily be added by a user as an additional postulate.) Time intervals are not included among the primitives of PSL Core, as intervals can be defined with respect to TIMEPOINTS and ACTIVITIES. TIMEDURATIONS are included in an extension of the PSL Core that builds upon [14].

The basic notions of the PSL Core are axiomatized formally as a first-order theory. These axioms simply capture, in a precise way, the basic properties of the PSL ontology. The basic axioms for ACTIVITIES, OBJECTs, and TIMEPOINTS are listed in Section 4.2 below.

3.4 Extensions in PSL 1.0

The set of extensions in PSL 1.0 fall roughly into three “families”:

- PSL “Outer Core”
- Generic Activities
- Schedules

3.4.1 PSL Outer Core

There is a small set of extensions that are so generic and pervasive in their applicability that we set them apart by calling them the PSL Outer Core. These three extensions are:

- Subactivity Extension
- Activity-Occurrence Extension
- States Extension

The Subactivity Extension describes how activities can be aggregated and decomposed. It also defines the concept of primitive activity, which can not be decomposed into any further activities. The Activity-Occurrence Extension defines relations that allow the description of how activity-occurrences relate to one another with respect to the time at which they start and end. The State Extension introduces the concept of state (before an activity-occurrence) and post-state (after an activity-occurrence).

3.4.2 Generic Activities and Ordering Relations

Figure 4 illustrates the modules in PSL that are required to define the terminology for generic classes of activities and their ordering relations. There are nine relevant extensions to PSL Core, four dealing with generic process modeling concepts and five dealing with schedules. The five focusing on schedules will be discussed in Section 3.4.3. The four dealing with generic process modeling concepts are:

- Ordering Relations
- Nondeterministic Activities
- Complex Sequence Ordering Relations
- Junctions

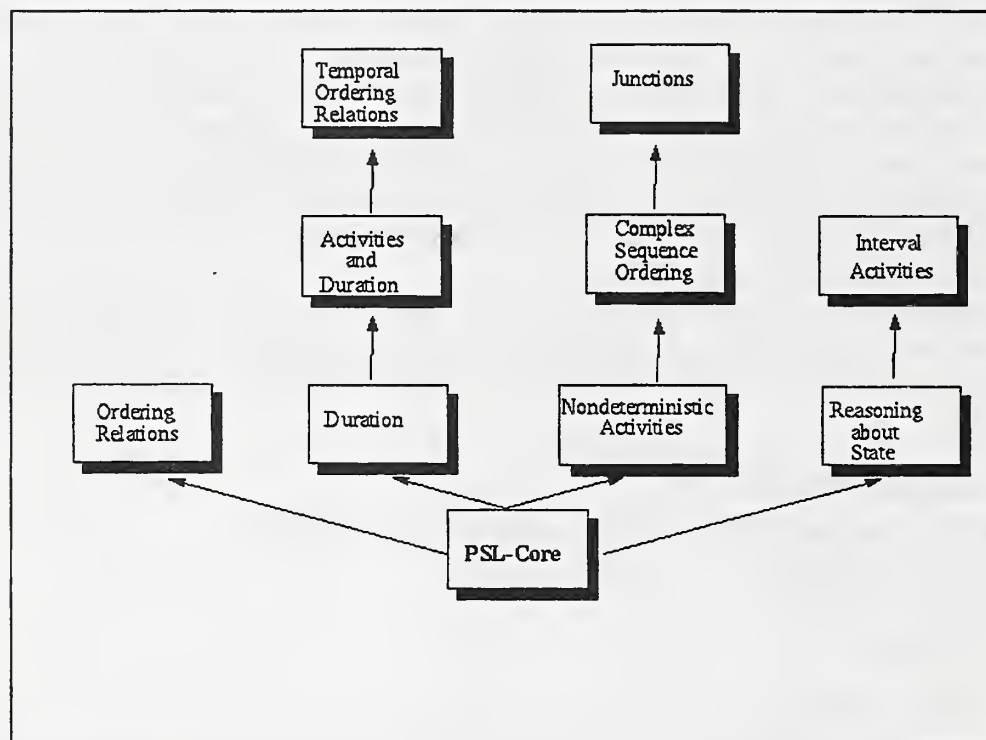


Figure 4: PSL modules for generic classes of activities and their ordering relations

The first of these extensions characterize deterministic activities. The final three extensions characterize nondeterministic activities in which not every subactivity occurs when the activity occurs; for example, to fabricate an engine block, one may either use the casting machine or modify an existing engine block. Junctions are a particular class of nondeterministic activities used to define notions such as splits and joins. Within a split, one of several activities may possibly occur next, whereas within a join, one of several activities must occur before the next activity occurs.

3.4.3 PSL Extensions for Schedules

These extensions were motivated by the applications in the PSL pilot implementation, in particular ILOG Scheduler 4.3. At the beginning of the pilot implementation of PSL,

there were no extensions capable of completely defining concepts such as temporal constraints. It was therefore necessary to design new extensions containing terminology whose definitions correctly and completely captured the intuitive meaning of the ILOG Schedule concepts.

Scheduling can be characterized intuitively as the assignment of resources to activities such that the temporal constraints are satisfied. Temporal constraints include the duration of activities and the temporal ordering of activity-occurrences. These intuitions lead to the introduction of five extensions within PSL 1.0, shown in Figure 4:

- Durations
- Activities and Duration
- Temporal Ordering Relations
- Reasoning about State
- Interval Activities

3.5 Approach for Developing Extensions

From the above list of extensions, one may see that certain representational areas within PSL have been thoroughly worked out and some have not been addressed yet. For example, the area relating to “ordering of activities” has been well addressed within the extensions of “Ordering Relations for Complex Sequence Actions,” “Ordering Relations over Activities,” and “Temporal Ordering.” However, other representational areas such as “Process Intent” have not yet been addressed.

The development of PSL has proceeded on an as-needed basis. The initial PSL ontology was developed using a single scenario, the EDAPS (Electromechanical Design and Planning System) scenario developed by Steve Smith at the University of Maryland [15]. The concepts introduced in that scenario were defined and modeled within PSL and later extended as other scenarios were explored. The PSL ontology was then further expanded to incorporate the concepts introduced in various manufacturing software applications when PSL was used to exchange process information among these packages. As more software applications become “PSL-compliant,” PSL will be continually expanded to ensure that ALL process-related concepts are capable of being represented within the language.

4 Informal Documentation

4.1 Introduction

The purpose of this section is to provide an informal documentation of the PSL Core and of its extensions. By *informal documentation* we mean a description in English of the intended meaning of the concepts introduced or defined in the PSL Core and its extensions. The formal definitions and axioms (expressed in the knowledge interchange format) that constitute the PSL Core and extensions are included in this document for reference purposes. The informal documentation is based solely and completely on the available PSL formal definitions and axioms.

The classification used in this document is the one provided by the IDEF5 Ontology Capture method. Briefly stated, *kinds* are similar to types or classes. They represent groups of things that share the same characteristics or properties. *Individuals* are similar to objects in the object-oriented paradigm. Each individual is uniquely identifiable and distinguishable from all other individuals. *Relations* are used to express relationships. Finally, *functions* are used to express properties.

4.2 PSL Core

This section provides both formal and informal documentation for the PSL Core.

4.2.1 Kinds for the PSL Core

The PSL Core introduces the following kinds of elements:.

Concept	Informal Definition
activity	A class or type of action. For example, 'paint-part' is an activity. It is the class of actions in which parts are being painted.
activity-occurrence	An event or action that takes place at a specific place and time. An instance or occurrence of an activity. E.g., paint-part is an activity, painting in Maryland at 2 PM on May 25, 1998 is an activity-occurrence.
timepoint	A point in time.
object	Anything that is not a timepoint or an activity.

The following axioms pertain to these four kinds.

- *Axiom 9. Everything is either an activity, an activity-occurrence, an object, or a timepoint.*
- *Axiom 10. Activities, activity-occurrences, objects, and timepoints are all distinct kinds of things.*

4.2.1.1 Activity

Activities are arguments to the following relations.

Relation	Arguments	Informal Definition
is-occurring-at	Activity-occurrence, timepoint	The activity corresponding to the specified activity occurrence is occurring at the specified timepoint. I.e., there exists an occurrence of the activity that is such that the specified timepoint is between or equal to the begin timepoint and end timepoint of the occurrence.
occurrence-of	activity-occurrence, activity	The activity-occurrence is a particular occurrence of the given activity.

4.2.1.2 Activity-Occurrence

Activity-occurrences are arguments to the following relations.

Relation	Arguments	Informal Definition
participates-in	object, activity, timepoint	The given object plays some (indeterminate) role in an occurrence of the given activity at the given timepoint.
occurrence-of	activity-occurrence, activity	The activity-occurrence is a particular occurrence of the given activity.

Activity-occurrences are arguments to the following functions.

Function	Arguments	Return value	Informal Definition
beginof	activity-occurrence	timepoint	The timepoint at which the occurrence begins.
endof	activity-occurrence	timepoint	The timepoint at which the occurrence ends.

The following axioms pertain to activity-occurrences.

- *Axiom 12. An activity-occurrence is the occurrence-of a single activity.*
- *Axiom 14. The timepoint at which an activity-occurrence begins always precedes the timepoint at which the activity-occurrence ends.*

4.2.1.3 Objects

Objects are arguments to the following relations.

Relation	Arguments	Informal Definition
exists-at	object, timepoint	The object exists at the given timepoint.
participates-in	object, activity-occurrence, time point	The given object plays some (indeterminate) role in the given activity occurrence at the given timepoint.

Objects are arguments to the following functions.

Function	Arguments	Return value	Informal Definition
beginof	object	timepoint	The timepoint at which the object comes into existence.
endof	object	timepoint	The timepoint at which the object ceases to exist.

4.2.1.4 Timepoint

Timepoints are arguments to the following relations.

Relation	Arguments	Informal Definition
participates-in	object, activity-occurrence, time point	The given object plays some (indeterminate) role in the given activity occurrence at the given timepoint.
before	timepoint, timepoint	This relation is used to impose a total ordering on timepoints.
between	timepoint, timepoint, timepoint	Strictly less than and strictly greater than.
beforeEq	timepoint, timepoint	Less or equal than.
betweenEq	timepoint, timepoint	Less or equal to and greater or equal to.
exists-at	object, timepoint	The object exists at the given timepoint.
is-occurring-at	Activity-occurrence, timepoint	The specified activity-occurrence is occurring at the specified timepoint. I.e., there exists an occurrence of an activity that is such that the specified timepoint is between or equal to the begin timepoint and end time point of the occurrence.

Timepoints are return values to the following functions.

Function	Arguments	Return value	Return value
Beginof	object	timepoint	The timepoint at which the object comes into existence.
Endof	object	timepoint	The timepoint at which the object ceases to exist.

4.2.2 Individuals for the PSL Core

The PSL Core introduces the following individuals.

inf-	The timepoint that is before all other timepoint.
inf+	The timepoint that is after all other timepoint.

The following axioms pertain to inf+ and inf-.

- *Axiom 5. Inf- is before all other timepoints.*
- *Axiom 6. Every timepoint else than inf+ is before inf+*
- *Axiom 7. Given any timepoint t other than inf-, there is a timepoint between inf- and t.*
- *Axiom 8. Given any timepoint t other than inf+, there is a timepoint between t and inf+.*

4.2.3 Primitive Relations for the PSL Core

The PSL Core introduces the following relations.

Relation	Arguments	Informal Definition
Before	timepoint, timepoint	This relation is used to impose a total ordering on timepoints.
occurrence-of	activity-occurrence, activity	The activity-occurrence is a particular occurrence of the given activity.
participates-in	object, activity-occurrence, timepoint	The given object plays some role in the given occurrence of an activity at the given timepoint.

4.2.3.1 Relation before

The following axioms pertain to the before relation.

- *Axiom 1. The before relation only holds between timepoints.*
- *Axiom 2. The before relation is a total ordering.*
- *Axiom 3. The before relation is non-reflexive.*
- *Axiom 4. The before relation is transitive.*

4.2.3.2 Relation occurrence-of

The following axioms pertain to the occurrence-of relation.

- *Axiom 11. The occurrence-of relation only holds between activities and activity-occurrences.*
- *Axiom 12. An activity-occurrence is the occurrence-of a single activity.*
- *Axiom 17. Every activity-occurrence is an occurrence-of an activity.*

4.2.3.3 Relation participates-in

The following axioms pertain to the participates-in relation.

- *Axiom 15. The participates-in relation only holds between objects, activities, and timepoints, respectively.*
- *Axiom 16. An object can participate in an activity only at those timepoints at which both the object exists and the activity is occurring.*

4.2.4 Primitive Functions for the PSL Core

The PSL Core introduces the following functions

Function	Arguments	Return type	Informal Definition
endof	activity	timepoint	The timepoint at which the activity ends.
endof	object	timepoint	The timepoint at which the object ceases to exist.
beginof	activity	timepoint	The timepoint at which the activity begins.
beginof	object	timepoint	The timepoint at which the object comes into existence.

The following axioms pertain to the beginof and endof functions.

- *Axiom 13. The begin and end of an activity-occurrence or object are timepoints.*
- *Axiom 14. The timepoint at which an activity-occurrence begins always precedes the timepoint at which the activity-occurrence ends.*

4.2.5 Defined Relations for the PSL Core

The PSL Core defines the following relations.

Relation	Arguments	Informal Definition
Between	timepoint, timepoint, timepoint	Strictly less than and strictly greater than.
BeforeEq	timepoint, timepoint	Less or equal than.
BetweenEq	timepoint, timepoint, timepoint	Less or equal to and greater or equal to.
exists-at	object, timepoint	A point in time in which an object exists.
is-occurring-at	Activity-occurrence, timepoint	The specified activity-occurrence is occurring at the specified timepoint. I.e., the specified timepoint is between or equal to the begin timepoint and end timepoint of the specified activity occurrence.

4.2.5.1 Relation between

The following is the formal definition for the *between* relation.

Definition 1. Timepoint q is between timepoints p and r if and only if p is before q and q is before r.

4.2.5.2 Relation beforeEq

The following is the formal definition for the *beforeEq* relation.

Definition 2. Timepoint p is beforeEq timepoint q if and only if p is before or equal to q.

4.2.5.3 Relation betweenEq

The following is the formal definition for the *betweenEq* relation.

Definition 3. Timepoint q is *betweenEq* timepoints p and r if and only if p is before or equal to q , and q is before or equal to r .

4.2.5.4 Relation exists-at

The following is the formal definition for the *exists-at* relation.

Definition 4. An object exists-at a timepoint p if and only if p is *betweenEq* its begin and end points.

4.2.5.5 Relation is-occurring-at

The following is the formal definition for the *is-occurring-at* relation.

Definition 5. An activity occurrence is-occurring-at a timepoint p if and only if p is *betweenEq* the activity occurrence's begin and end points.

4.2.6 Definitions and Axioms for the PSL Core in the formal language

In this section the definitions and axioms for the PSL Core are given explicitly in the formal PSL language. (Note: The lexicon items 'defrelation', 'exists', 'forall', 'and', 'or', 'not', '=', '<=>', and '=>' are defined in the KIF Reference Manual [3].)

Definition 1. Timepoint q is between timepoints p and r if and only if p is before q and q is before r .

```
(defrelation between (?p ?q ?r) :=
  (and (before ?p ?q) (before ?q ?r)))
```

Definition 2. Timepoint p is *beforeEq* timepoint q if and only if p is before or equal to q .

```
(defrelation beforeEq (?p ?q) :=
  (and (timepoint ?p) (timepoint ?q)
    (or (before ?p ?q)
      (= ?p ?q))))
```

Definition 3. Timepoint q is *betweenEq* timepoints p and r if and only if p is before or equal to q , and q is before or equal to r .

```
(defrelation betweenEq (?p ?q ?r) :=
  (and (beforeEq ?p ?q)
    (beforeEq ?q ?r)))
```

Definition 4. An object exists-at a timepoint p if and only if p is *betweenEq* its begin and end points.

```
(defrelation exists-at (?x ?p) :=
  (and (object ?x)
    (betweenEq (beginof ?x) ?p (endof ?x))))
```

Definition 5. An activity occurrence is-occurring-at a timepoint p if and only if p is betweenEq the activity occurrence's begin and end points.

```
(defrelation is-occurring-at (?occ ?p) :=
  (and (activity-occurrence ?occ)
    (betweenEq (beginof ?occ) ?p (endof ?occ))))
```

4.2.7 PSL Core Axioms

Axiom 1. The before relation only holds between timepoints.

```
(forall (?p ?q)
  (=> (before ?p ?q)
    (and (timepoint ?p)
      (timepoint ?q))))
```

Axiom 2. The before relation is a total ordering.

```
(forall (?p ?q)
  (=> (and (timepoint ?p)
    (timepoint ?q))
    (or (= ?p ?q)
      (before ?p ?q)
      (before ?q ?p))))
```

Axiom 3. The before relation is non-reflexive.

```
(forall (?p)
  (not (before ?p ?p)))
```

Axiom 4. The before relation is transitive.

```
(forall (?p ?q ?r)
  (=> (and (before ?p ?q)
    (before ?q ?r))
    (before ?p ?r)))
```

Axiom 5. Inf- is before every other timepoint.

```
(forall (?t)
  (=> (timepoint ?t)
    (beforeEq inf- ?t)))
```

Axiom 6. Every timepoint else than inf+ is before inf+

```
(forall (?t)
  (=> (timepoint ?t)
    (beforeEq ?t inf+)))
```

Axiom 7. Given any timepoint t other than inf-, there is a timepoint between inf- and t .

```
(forall (?t)
```

```

(=> (and (timepoint ?t)
         (not (= ?t inf-)))
     (exists (?u)
              (between inf- ?u ?t))))

```

Axiom 8. Given any timepoint t other than inf- , there is a timepoint between t and inf- .

```

(forall (?t)
  (=> (and (timepoint ?t)
           (not (= ?t inf+)))
      (exists (?u)
               (between ?t ?u inf+)))))

```

Axiom 9. Everything is either an activity, an activity-occurrence, an object, or a timepoint.

```

(forall (?x)
  (or (activity ?x)
      (activity-occurrence ?x)
      (object ?x)
      (timepoint ?x)))

```

Axiom 10. Activities, activity-occurrences, objects, and timepoints are all distinct kinds of things.

```

(forall (?x)
  (and (=> (activity ?x)
          (not (or (activity-occurrence ?x)
                  (object ?x)
                  (timepoint ?x))))
      (=> (activity-occurrence ?x)
          (not (or (object ?x)
                  (timepoint ?x))))
      (=> (object ?x)
          (not (timepoint ?x)))))

```

Axiom 11. The occurrence-of relation only holds between activities and activity-occurrences.

```

(forall (?a ?occ)
  (=> (occurrence-of ?occ ?a)
      (and (activity ?a)
            (activity-occurrence ?occ))))

```

Axiom 12. An activity-occurrence is the occurrence-of a single activity.

```

(forall (?occ ?a1 ?a2)
  (=> (and (occurrence-of ?occ ?a1)
           (occurrence-of ?occ ?a2))
      (= ?a1 ?a2)))

```

Axiom 13. The begin and end of an activity-occurrence or object are timepoints.

```

(forall (?a ?x)

```

```
(=> (or (occurrence-of ?x ?a)
        (object ?x))
     (and (timepoint (beginof ?x))
           (timepoint (endof ?x)))))
```

Axiom 14. The timepoint at which an activity-occurrence begins always precedes the timepoint at which the activity-occurrence ends.

```
(forall (?a ?x)
  (=> (or (occurrence-of ?x ?a)
          (object ?x))
      (beforeEq (beginof ?x) (endof ?x)))))
```

Axiom 15. The participates-in relation only holds between objects, activities, and timepoints, respectively.

```
(forall (?x ?occ ?t)
  (=> (participates-in ?x ?occ ?t)
      (and (object ?x)
            (activity-occurrence ?occ)
            (timepoint ?t))))
```

Axiom 16. An object can participate in an activity only at those timepoints at which both the object exists and the activity is occurring.

```
(forall (?x ?occ ?t)
  (=> (participates-in ?x ?occ ?t)
      (and (exists-at ?x ?t)
            (is-occurring-at ?occ ?t))))
```

4.3 Subactivity Extension

The purpose of this extension is to define the subactivity relation, which specifies how activities can be aggregated and decomposed. It also defines the concept of primitive activity, which cannot be decomposed into any further activities.

4.3.1 Defined Classes in the Subactivity Extension

The subactivity extension defines the following kind.

Kind	Informal Definition
primitive-activity	A primitive activity is an activity that does not have any subactivities.

4.3.2 Defined Relations in the Subactivity Extension

The subactivity extension defines the following relation.

Relation	Arguments	Informal Definition
subactivity	activity, activity	This relation defines a partial ordering over the set of activities with respect to aggregation and decomposition

4.3.3 Formal Axioms in the Subactivity Extension

Axiom 1. The subactivity relation is reflexive.

(forall (?a) (subactivity ?a ?a))

Axiom 2. The subactivity relation is asymmetric.

(forall (?a1 ?a2)
 (=> (and (subactivity ?a1 ?a2)
 (subactivity ?a2 ?a1))
 (= ?a1 ?a2)))

Axiom 3. The subactivity relation is transitive.

(forall (?a1 ?a2 ?a3)
 (=> (and (subactivity ?a1 ?a2)
 (subactivity ?a2 ?a3))
 (subactivity ?a1 ?a3)))

Axiom 4. For any two activities, there exists another activity which contains them both as subactivities.

(forall (?a1 ?a2)
 (exists (?a3)
 (and (subactivity ?a1 ?a3)
 (subactivity ?a2 ?a3))))

Definition 1. A primitive activity is an activity that does not have any proper subactivities.

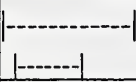
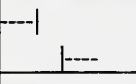
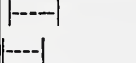
(defrelation primitive-activity (?a) :=
 (forall (?a1)
 (=> (subactivity ?a1 ?a)
 (= ?a1 ?a))))

4.4 Activity-Occurrences Extension

The purpose of this extension is to define relations that allow the description of how activity-occurrences relate to one another with respect to the time at which they start and end.

4.4.1 Introduced Relations in the Activity-Occurrences Extension

The activity-occurrence extension introduces the following relations.

Relation	Arguments	Informal Definition
occurrence-contains 	activity-occurrence, activity-occurrence	An occurrence occ1 contains another occ2 if occ1 happens during occ2.
occurrence-earlier 	activity-occurrence, activity-occurrence	An occurrence occ1 is earlier than another occ2 if occ1 ends before occ2 begins.
occurrence-overlap 	activity-occurrence, activity-occurrence	Two occurrences occ1 and occ2 overlap if there is an interval of time during which both occurrences are occurring and if occ2 starts before occ1 starts and ends before occ1 ends.

The following axioms pertain to these relations.

Axiom 1. If two occurrences stand in the successor relation, than they stand in the occurrence-earlier relation.

Axioms 2 and 3. The relations *occurrence-contains* and *occurrence-overlap* are reflexive.

Axiom 4. If an activity-occurrence stands in the relation *occurrence-earlier* with itself, than its beginning and ending point are equal.

Axiom 7. If an activity-occurrence occ1 contains an activity-occurrence occ2, than the beginning timepoint of occ1 is before or equal to the beginning timepoint of occ2, and the ending timepoint of occ2 is before or equal to the ending timepoint of occ1.

Axiom 8. If an activity-occurrence occ1 is earlier than an activity-occurrence occ2, then the beginning timepoint of occ1 is before or equal to the beginning timepoint of occ2.

Axiom 9. If an activity-occurrence occ1 overlaps with an activity-occurrence occ2, then the beginning timepoint of occ2 is before or equal to the beginning timepoint of occ1, the beginning timepoint of occ1 is before or equal to the ending timepoint of occ2 , and the ending timepoint of occ2 is before or equal to the ending timepoint of occ1.

4.4.2 Defined Relations in the Activity-Occurrences Extension

The activity-occurrence extension introduces the following relations.

Relation	Arguments	Informal Definition
successor	activity-occurrence, activity-occurrence, activity-occurrence	An activity-occurrence occ2 is the successor of an activity-occurrence occ1 if occ1 occurs earlier than occ2 and if they are no activity occurrence that occurs between occ1 and occ2.
subactivity-occurrence	activity-occurrence, activity-occurrence	An activity-occurrence occ1 is a subactivity occurrence of an activity-occurrence occ2 if the activity of which occ1 is an activity-occurrence is a subactivity of the activity of which occ2 is an activity-occurrence and occ1 stands in the occurrence-contains relation with occ2.

4.4.3 Formal Axioms in the Activity-Occurrences Extension

Definition 1. One activity-occurrence is the successor of another if and only if the first activity-occurrence is earlier and there does not exist any other activity-occurrence between them.

```
(defrelation successor (?occ1 ?occ2) :=
  (and (occurrence-earlier ?occ1 ?occ2)
    (not (exists (?occ3)
      (occurrence-earlier ?occ1 ?occ3)
      (occurrence-earlier ?occ3 ?occ2))))))
```

Definition 2. An activity-occurrence occ1 is a subactivity-occurrence of an activity-occurrence occ2 if the activity of which occ1 is an activity-occurrence is a subactivity of the activity of which occ2 is an activity-occurrence and occ1 stands in the occurrence-contains relation with occ2 .

```
(defrelation subactivity-occurrence (?occ1 ?occ2) :=
  (and (forall (?a1 ?a2)
    (=> (and (occurrence-of ?occ1 ?a1)
      (occurrence-of ?occ2 ?a2))
      (subactivity ?a1 ?a2)))
    (occurrence-contains ?occ2 ?occ1)))
```

Axiom 1. If two activity-occurrences stand in the successor relation, than they stand in the occurrence-earlier relation.

```
(forall (?occ1 ?occ2)
  (=> (successor ?occ1 ?occ2)
    (occurrence-earlier ?occ1 ?occ2)))
```

Axioms 2 and 3. The relations *occurrence-contains* and *occurrence-overlap* are reflexive.

```
(forall (?occ) (occurrence-contains ?occ ?occ))
```

```
(forall (?occ) (occurrence-overlap ?occ ?occ))
```

Axiom 4. If an activity-occurrence stand in the relation *occurrence-earlier* with itself, than its beginning and ending point are equal.

```
(forall (?occ)
  (=> (occurrence-earlier ?occ ?occ)
    (= (beginof ?occ) (endof ?occ))))
```

Axioms 5. The relations *occurrence-contains* is transitive.

```
(forall (?occ1 ?occ2 ?occ3)
  (=> (and (occurrence-contains ?occ1 ?occ2)
    (occurrence-contains ?occ2 ?occ3))
    (occurrence-contains ?occ1 ?occ3)))
```

Axioms 6. The relations *occurrence-earlier* is transitive.

```
(forall (?occ1 ?occ2 ?occ3)
  (=> (and (occurrence-earlier ?occ1 ?occ2)
    (occurrence-earlier ?occ2 ?occ3))
    (occurrence-earlier ?occ1 ?occ3)))
```

Axiom 7. If an activity-occurrence *occ1* contains an activity-occurrence *occ2*, than the beginning timepoint of *occ1* is before or equal to the beginning timepoint of *occ2*, and the ending timepoint of *occ2* is before or equal to the ending timepoint of *occ1*.

```
(forall (?occ1 ?occ2)
  (=> (occurrence-contains ?occ1 ?occ2)
    (and (beforeEq (beginof ?occ1) (beginof ?occ2))
      (beforeEq (endof ?occ2) (endof ?occ1)))))
```

Axiom 8. If an activity-occurrence *occ1* is earlier than an activity-occurrence *occ2*, then the ending timepoint of *occ1* is before or equal to the beginning timepoint of *occ2*.

```
(forall (?occ1 ?occ2)
  (=> (occurrence-earlier ?occ1 ?occ2)
    (beforeEq (endof ?occ1) (beginof ?occ2))))
```

Axiom 9. If an activity-occurrence *occ1* overlaps with an activity-occurrence *occ2*, then the beginning timepoint of *occ2* is before or equal to the beginning timepoint of *occ1*, the beginning timepoint of *occ1* is before or equal to the ending timepoint of *occ2*, and the ending timepoint of *occ2* is before or equal to the ending timepoint of *occ1*.


```

(forall (?occ1 ?occ2)
  (=> (occurrence-overlap ?occ1 ?occ2)
    (and (beforeEq (beginof ?occ2) (beginof ?occ1))
      (beforeEq (beginof ?occ1) (endof ?occ2))
      (beforeEq (endof ?occ2) (endof ?occ1))))))

```

4.5 States Extension

The extension introduces the concepts of *state* (before an activity-occurrence) and *post-state* (after an activity-occurrence).

4.5.1 Classes of Objects in the States Extension

The following kind is defined in the state extension.

Kind	Informal Definition
Fluent	A fluent is a property of the world that can change as a result of an activity occurring. A fluent is said to hold before an activity-occurrence if the world had that property before the activity-occurrence and to hold after an activity-occurrence if the world has that property after the activity-occurrence.

4.5.2 Introduced Relations in the States Extension

The states extension introduces the following relations.

Relation	Arguments	Informal Definition
State	fluent, activity-occurrence	The state relation holds between a fluent and an activity-occurrence, if the fluent holds before the activity-occurrence.
Post-state	fluent, activity-occurrence	The effect relation holds between a fluent and an activity if the fluent holds after the activity-occurrence.

4.6 Integer and Duration Extension

The primary purpose of this extension is to axiomatize the concept of a timeduration and the auxiliary notion of integer. Intuitively, a timeduration is a measure of the “temporal distance” between two points. Thus, in addition to a new predicate *Timeduration*, we also introduce a new function *duration* and a new function symbol ‘Duration’. The *duration* function takes two points as arguments. Intuitively, these two points represent

the start and end points of the interval from the first argument to the second. Note that it turns out to be most convenient not to let *duration* be commutative. That is, the duration from t to u is not the duration from u to t , unless $t = u$, in which case the duration from t to u is *zero* (the “null” duration). Intuitively, then, the duration from point t to point u is a vector that has a negative or positive direction depending on whether or not t is before u . Durations can be added to each other, and multiplied by integers. Because there are points at infinity, we need to allow for infinite durations (e.g., the duration from any finite point to a point at infinite). For this purpose, in addition to a special *zero* duration, we introduce an infinite negative duration *max-* and an infinite positive duration *max+*.

We will present the integer portion of the extension first, as the *timeduration* extension builds upon it.

4.6.1 Primitive Kinds in the Integer Extension

Concept	Informal Definition
Integer	The class of integers: 0, -1, 1, -2, 2, etc.

4.6.2 Defined Kinds in the Integer Extension

Kind	Informal Definition
PosInt	The class of positive integers: 1, 2, 3, etc.
NegInt	The class of negative integers: -1, -2, -3, etc.

4.6.3 Individuals in the Integer Extension

The integer extension introduces the following individuals.

Individual	Informal Definition
0	The integer 0

The following axioms pertain to this individual.

Axiom 8. The sum of any integer and 0 is that integer.

Axiom 10. The difference between any integer and 0 is that integer.

4.6.4 Functions in the Integer Extension

Integers are arguments to the following functions.

Function	Arguments	Return Type	Informal Definition
+1	Integer	Integer	+1 is the successor function on integers

-1	Integer	Integer	-1 is the predecessor function on integers
+	Integer, Integer	Integer	+ is the addition function
-	Integer, Integer	Integer	- is the subtraction function

The following axioms pertain to these functions.

Axiom 1. The successor and predecessor functions are one-to-one on the integers.

Axiom 6. An integer is less than its successor and greater than its predecessor.

Axiom 7. No integer is between a given integer and its successor or predecessor.

Axiom 9. The successor of any integer i with the successor of any integer j is the successor of the sum of i and j ; the successor of any integer i with the predecessor of any integer j is the predecessor of the sum of i and j .

Axiom 11. The difference between an integer i and the successor of any integer j is the predecessor of the difference between i and j ; the difference between an integer i and the predecessor of any integer j is the successor of the difference between i and j .

4.6.5 Relations on Integers

Integers are arguments to the following relation.

Relation	Arguments	Informal Definition
<	Integer, Integer	< is the less-than relation on integers

The following axioms pertain to this relation.

Axiom 3. The less-than relation is transitive on the integers.

Axiom 4. The less-than relation is irreflexive on the integers.

Axiom 5. The less-than relation is a total ordering on the integers.

Axiom 6. An integer is less than its successor and greater than its predecessor.

Axiom 7. No integer is between a given integer and its successor or predecessor.

4.6.6 Formal Definitions and Axioms for Integers

In order to give a theory of the integers we introduce the predicate `Integer`, the constant `'0'`, the function symbols `' +1 '` and `' -1 '` indicating the successor and predecessor functions, respectively, the predicate `' < '` indicating the less-than relation on the integers, the addition and subtraction symbols `' + '`, and `' - '`, and the following definitions and axioms.²

4.6.6.1 Definitions for Integers

Definition 1. A *positive integer* is an integer that is greater than 0.

```
(defrelation Posint (?i) :=  
  (and (Integer ?i) (< 0 ?i)))
```

Definition 2. A *negative integer* is an integer that is less than 0.

```
(defrelation Negint (?i) :=  
  (and (Integer ?i) (< ?i 0)))
```

It is convenient to introduce definitions for the traditional numerals, i.e.,

```
(defobject 1 := (+1 0))
```

```
(defobject 2 := (+1 1))
```

and so on. It is also convenient to use the string `"-τ,"` for any term τ to abbreviate `"(- 0 τ)." (This can't be considered a formal definition because strings beginning with " - " are not constants according to the BNF.)`

² Thanks are due to Tom Costello of Stanford University for providing an axiomatization of the integers upon which the axioms in this document were based.

4.6.6.2 Axioms for Integers

Axiom 1. The successor and predecessor functions are one-to-one on the integers.

```
(forall (?i ?j : (Integer ?i) (Integer ?j))  
  (and (=> (= (+1 ?i) (+1 ?j))
```

Axiom 2. The successor and predecessor functions are one-to-one on the integers.

```
(forall (?i ?j : (Integer ?i) (Integer ?j))  
  (and (=> (= (+1 ?i) (+1 ?j))  
        (= ?i ?j))  
    (=> (= (-1 ?i) (-1 ?j))  
        (= ?i ?j)))
```

Axiom 3. The less-than relation is transitive on the integers.

```
(forall (?i ?j ?k : (Integer ?i) (Integer ?j) (Integer ?k))  
  (=> (and (< ?i ?j) (< ?j ?k))  
      (< ?i ?k)))
```

Axiom 4. The less-than relation is irreflexive on the integers.

```
(forall ?i (not (< ?i ?i)))
```

Axiom 5. The less-than relation is a total ordering on the integers.

```
(forall (?i ?j : (Integer ?i) (Integer ?j))  
  (or (< ?i ?j) (< ?j ?i) (= ?i ?j)))
```

Axiom 6. An integer is less than its successor and greater than its predecessor.

```
(forall (?i : (Integer ?i))  
  (and (< ?i (+1 ?i))  
      (< (-1 ?i) ?i)))
```

Axiom 7. No integer is between a given integer and its successor or predecessor.

```
(forall (?i ?j : (Integer ?i) (Integer ?j))  
  (not (or (and (< ?i ?j) (< ?j (+1 ?i))  
                (and (< ?j ?i) (< (-1 ?i) ?j)))))
```

Axiom 8. The sum of any integer and 0 is that integer.

```
(forall (?i : (Integer ?i))  
  (= (+ ?i 0) ?i))
```

Axiom 9. The successor of any integer i with the successor of any integer j is the successor of the sum of i and j ; the successor of any integer i with the predecessor of any integer j is the predecessor of the sum of i and j .

```
(forall (?i ?j : (Integer ?i) (Integer ?j))
  (and (= (+ ?i (+1 ?j)) (+1 (+ ?i ?j)))
    (= (+ ?i (-1 ?j)) (-1 (+ ?i ?j)))))
```

Axiom 10. The difference between any integer and 0 is that integer.

```
(forall (?i : (Integer ?i))
  (= (- ?i 0) 0))
```

Axiom 11. The difference between an integer i and the successor of any integer j is the predecessor of the difference between i and j ; the difference between an integer i and the predecessor of any integer j is the successor of the difference between i and j .

```
(forall (?i ?j : (Integer ?i) (Integer ?j))
  (and (= (- ?i (+1 ?j)) (-1 (- ?i ?j)))
    (= (- ?i (-1 ?j)) (+1 (- ?i ?j)))))
```

4.6.6.3 Induction

The above axioms are sufficient for doing basic integer arithmetic. If more general properties of the integers are needed (e.g., if one wished to show not simply that $(= (+ 5 -5) 0)$ but that $(\text{forall } ?i (= (+ ?i -?i) 0))$ generally, we should need also some form of induction schema. The most general first-order induction schema is the following; let ϕ be a sentence with the variable v free:

```
(=> (and ( $\phi[v/0]$ 
  (forall (?i : (Integer ?i))
    ( $\Rightarrow \phi[v/?i] \phi[v/(+1 ?i)]$ )))
  (forall (?i : (Integer ?i))
    ( $\Rightarrow \phi[v/?i] \phi[v/(-1 ?i)]$ )))
  (forall (?i : (Integer ?i))  $\phi[v/?i]$ ))
```

That is, roughly, any property true of 0 and true of both the predecessor and successor of i when it is true of i , for any integer i , is true of all integers. Since basic integer arithmetic is all we will need for most purposes, we will not include any instances of the induction schema among our core axioms.

To be able to give local conditions involving integers, we specify that:

Axiom 12. Every integer participates in every activity.

```
(=> (and (Integer ?i) (activity ?a))
  (In ?i ?a))
```

This condition simply enables us to use the “Holds” predicate below with respect to sentences that refer to integers, and should not be thought of as capturing some deep insight about the nature of activities.

4.6.7 Primitive Kinds in the Timedurations Extension

The duration extension introduces the following primitive kind.

Concept	Informal Definition
Timeduration	The class of timedurations, intuitively, the possible lengths of time that an activity occurrence can last, represented simply as the pair of points that at which an activity occurrence begins and ends, respectively.

4.6.8 Individuals in the Timeduration Extension

The timeduration extension introduces the following individuals.

Individual	Informal Definition
zero	The instantaneous duration
max+	The maximum positive duration
max-	The maximum negative duration

The following axioms pertain to these individuals.

Axiom 13: zero, max+, and max- are all timedurations.³

Axiom 17. The result of *adding* any duration d to the duration *zero* is d .

Axiom 18. The sum of the duration from t to u and the duration from u to t is *zero*.

Axiom 19. The result of *adding* any duration other than *max+* to *max-* is *max-*, and vice versa.

Axiom 25. The duration from t to u is *zero* if and only if t and u are the same timepoint.

Axiom 26. The duration from any point other than *inf-* to *inf-* is *max-* and from any point other than *inf+* to *inf+* is *max+*.

Axiom 27. The duration from *inf-* to any point other than *inf-* is *max+* and from *inf+* to any point other than *inf+* is *max-*.

4.6.9 Defined Properties and Relations in the Duration Extension

The duration extension defines the following properties and relations.

³ *zero* is not necessarily to be identified with the number 0.

Relation	Arguments	Informal Definition
Positive	Timeduration	A timeduration is <i>positive</i> if, whenever it is the value of the duration function applied to points t and u , respectively, t is before u .
Negative	Timeduration	A timeduration is <i>negative</i> if, whenever it is the value of the duration function applied to points t and u , respectively, u is before t .
Shorter	Timepoint, Timepoint	One timeduration x is shorter than another z if and only if z is the result of adding some positive timeduration to x .

4.6.10 Defined Functions in the Duration Extension

The duration extension defines the following function.

Function	Arguments	Return Type	Informal Definition
duration-of	Object	Timeduration	The duration-of an object x is the value of the duration function applied to its begin and end points.

4.6.11 Functions in the Duration Extension

The duration extension introduces the following functions.

Function	Arguments	Return Type	Informal Definition
add	Timeduration, Timeduration	Timeduration	The <i>add</i> function yields the “duration sum” of two timedurations
mult	Integer, Timeduration	Timeduration	The <i>mult</i> function applied to an integer n and timeduration d yields the “duration product” of the two, intuitively, a duration n times “longer” than d .
duration	Timepoint, Timepoint	Timeduration	The <i>duration</i> function applied to two timepoints yields the timeduration that separates them.

The following axioms pertain to these functions.

Axiom 17. The result of *adding* any duration d to the duration *zero* is d .

Axiom 18. The sum of the duration from t to u and the duration from u to t is *zero*.

Axiom 19. The sum of any duration other than $max+$ and $max-$ is $max-$, and the result of adding any duration other than $max-$ to $max+$ is $max+$.

Axiom 20. The product of an integer with a timeduration is a timeduration.

Axiom 21. The product of the sum of two integers i and j and a duration d is the duration sum of the duration products of i and d and j and d .⁴

Axiom 22. The *duration* function maps two points – intuitively, the “interval” between them – to a timeduration.⁵

Axiom 23. Every timeduration is the duration of two points.

Axiom 24. Given a point i other than $inf-$ or $inf+$, the duration from i to any point is unique, i.e., it differs from the duration from i to any *other* point.

Axiom 25. The duration from t to u is zero iff t and u are the same point.

Axiom 26. The duration from any point other than $inf-$ to $inf-$ is $max-$ and from any point other than $inf+$ to $inf+$ is $max+$.

Axiom 27. The duration from $inf-$ to any point other than $inf-$ is $max+$ and from $inf+$ to any point other than $inf+$ is $max-$.

Axiom 28. The duration of an “interval” exactly comprising two “adjacent” intervals is the sum of the durations of the adjacent intervals.

Axiom 29. The result of multiplying -1 and the duration from t to u is the “inverse” of that duration, i.e., the duration from u to t .

4.6.11.1 Definitions for Timedurations

Definition 1. A timeduration is *positive* if, whenever it is the value of the duration function applied to points t and u , respectively, t is before u .

```
(defrelation Positive (?d) :=
  (and (Timeduration ?d)
    (forall (?t ?u : (Point ?t) (Point ?u))
      (=> (= ?d (Duration ?t ?u))
        (Before ?t ?u))))))
```

Definition 2. A timeduration is *negative* if, whenever it is the value of the duration function applied to points t and u , respectively, u is before t .

⁴ Note that, although *mult* is a total function (like all functions in first-order logic) we are only concerned with the case where the first argument of the function is a natural number n and the second argument is a timeduration d . In this case, *mult* yields a timeduration as value, intuitively, the duration that is n times as long as d .

⁵ We don't care what the duration function does when handed other objects. Note, again, that intervals are vectors, in that, if t is not u , the duration from t to u is not the same as the duration from u to t .

```

(defrelation Negative (?d) :=
  (and (Timeduration ?d)
    (forall (?t ?u : (Point ?t) (Point ?u))
      (=> (= ?d (Duration ?t ?u))
        (Before ?u ?t))))))

```

Definition 3. The *duration-of* an object x is the value of the duration function applied to its begin and end points.⁶

```

(deffunction duration-of (?x) :=
  (Duration (Beginof ?x) (Endof ?x)))

```

Definition 4. One timeduration x is *shorter than* another z iff z is the result of Adding some positive timeduration to x .

```

(defrelation Shorter (?d ?e) :=
  (exists! (?f) (and (Timeduration ?f)
    (Positive ?f)
    (= (add ?d ?e) ?f))))

```

4.6.11.2 Axioms for Timedurations

Timedurations are taken to be properties of pairs of timepoints. Timedurations will be defined for activities and objects (and, trivially, timepoints) in terms of their begin and end points. Timedurations can be compared, added together, and multiplied by integers, as clocks measure duration by counting.

Axiom 14: zero, max+, and max- are all timedurations.⁷

```

(and (Timeduration zero)
  (Timeduration max+)
  (Timeduration max-))

```

Axiom 15. The *add* function is symmetric and associative.

```

(forall (?d ?e ?f : (Timeduration ?d)
  (Timeduration ?e)
  (Timeduration ?f))
  (= (add ?d ?e) (add ?e ?d))
  (= (add ?d (add ?e ?f)) (add (add ?d ?e) ?f))))

```

Axiom 16. The result of adding two timedurations is a timeduration.

```

(forall (?d ?e : (Timeduration ?d) (Timeduration ?e))
  (Timeduration (add ?d ?e)))

```

Axiom 17. The result of adding any duration d to zero is d .

⁶ Note that a consequence of this definition and the axioms for *Beginof* and *Endof* is that timepoints have a duration-of zero, as they intuitively should.

⁷ zero is not necessarily to be identified with the number 0.

```
(forall (?d : (Timeduration ?d))
  (= (add zero ?d) ?d))
```

Axiom 18. The sum of the duration from t to u and the duration from u to t is zero.

```
(forall (?t ?u : (Point ?t) (Point ?u))
  (= zero (add (duration ?t ?u) (duration ?u ?t))))
```

Axiom 19. The result of adding any duration other than $max+$ to $max-$ is $max-$, and the result of adding any duration other than $max-$ to $max+$ is $max+$.

```
(forall (?d : (Timeduration ?d))
  (and (=> (not (= ?d max+))
    (= (add ?d max-) max-))
    (=> (not (= ?d max-))
    (= (add ?d max+) max+))))
```

Axiom 20. The product of an integer with a timeduration is a timeduration.

```
(forall (?i ?d)
  (=> (Integer ?i) (Timeduration ?d)
    (Timeduration (Mult ?i ?d))))
```

Axiom 21. The product of the sum of two integers i and j and a duration d is the duration sum of the duration products of i and d and j and d .⁸

```
(forall (?i ?d ?j : (Integer ?i) (Integer ?j) (Timeduration ?d))
  (and (= (mult (+ ?i ?j) ?d)
    (add (mult ?i ?d) (mult ?j ?d)))))
```

Axiom 22. The *duration* function maps two points – intuitively, the “interval” between them – to a timeduration.⁹

```
(forall (?t ?u : (Point ?t) (Point ?u))
  (Timeduration (duration ?t ?u)))
```

Axiom 23. Every timeduration is the duration of two points.

```
(forall (?d : (Timeduration ?d))
  (exists (?t ?u : (Point ?t ?u))
    (= ?d (duration ?t ?u))))
```

Axiom 24. Given a point i other than $inf-$ or $inf+$, the duration from i to any point is unique, i.e., it differs from the duration from i to any *other* point.

```
(forall (?i ?j ?k : (Point ?i) (Point ?j) (Point ?k))
```

⁸ Note that, although *mult* is a total function (like all functions in first-order logic) we are only concerned with the case where the first argument of the function is a natural number n and the second argument is a timeduration d . In this case, *mult* yields a timeduration as value, intuitively, the duration that is n times as long as d .

⁹ We don't care what the duration function does when handed other objects. Note, again, that intervals are vectors, in that, if t is not u , the duration from t to u is not the same as the duration from u to t .

```

(not (= ?i inf-)) (not (= ?i inf+)))
(=> (and (= (duration ?i ?j) (duration ?i ?k)))
    (= ?j ?k)))

```

Axiom 25. The duration from t to u is zero if and only if t and u are the same point.

```

(forall (?t ?u : (Point ?t) (Point ?u))
  (<=> (= (duration ?t ?u) zero)
    (= ?t ?u)))

```

Axiom 26. The duration from any point other than inf^- to inf^- is max^- and from any point other than inf^+ to inf^+ is max^+ .

```

(forall (?t : (Point ?t))
  (and (=> (not (= ?t inf-)) (= (duration ?t inf-) max-))
    (=> (not (= ?t inf+)) (= (duration ?t inf+) max+))))

```

Axiom 27. The duration from inf^- to any point other than inf^- is max^+ and from inf^+ to any point other than inf^+ is max^- .

```

(forall (?t : (Point ?t))
  (and (=> (not (= ?t inf-)) (= (duration inf- ?t) max+))
    (=> (not (= ?t inf+)) (= (duration inf+ ?t) max-))))

```

Axiom 28. The duration of an “interval” exactly comprising two “adjacent” intervals is the sum of the durations of the adjacent intervals.

```

(forall (?t ?u ?v: (Point ?t) (Point ?u) (Point ?v))
  (=> (and (Before ?t ?u) (Before ?u ?v)
    (= (duration ?t ?v) (add (duration ?t ?u) (duration ?u ?v)))))

```

Axiom 29. The result of multiplying -1 and the duration from t to u is the “inverse” of that duration, i.e., the duration from u to t .

```

(= (Mult -1 (duration ?t ?u)) (duration ?u ?t))

```

4.7 Ordering Relations over Activities Extension

The purpose of this extension is to provide relations to express temporal precedence relations among activities and activity-occurrences. The relations can only be used to talk about activity-occurrences that have occurred or will occur.

4.7.1 Classes of Activities in the Ordering Relations Extension

Kind	Informal Definition
poset-activity	A poset-activity is an activity that has a partially ordered set of primitive subactivities.
complex-poset-	A complex-poset-activity is an activity that has a partially ordered set of subactivities. These

activity	subactivities may be nondeterministic, leading to branches and junctions within the process flow
----------	--

4.7.2 Relations in the Ordering Relations Extension

The ordering relations over activities extension defines the following relations.

Relation	Arguments	Informal Definition
subactivity-precedes ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(subactivity-precedes occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the beginning timepoint of occ1 is earlier than the start ing timepoint of occ2.
next-activity	activity-occurrence, activity-occurrence, activity-occurrence	(next-activity occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that occ1 directly precedes occ2. That is, occ1 precedes occ2 and no other subactivity-occurrence of occ3 is such that occ1 precedes it and it precedes occ2.
initial-activity	activity, activity	An activity a1 is an initial activity of an activity a2 if and only if, whenever a1 occurs as a subactivity of a2, it is not the next activity of any other activity.
final-activity	activity, activity	An activity a1 is a final activity of an activity a2 if and only if, whenever a1 occurs as a subactivity of a2, no activity is its next activity.

The formal definitions for the relations in Ordering Relations over Activities Extension will be completed for the next release.

4.8 Ordering Relations For Complex Sequences of Activities Extension

The purpose of this extension is to provide relations to express temporal precedence relations among activities and among activity-occurrences.

4.8.1 Defined Relations in the Ordering Relations For Complex Sequences Extension

The ordering relation extension defines the following relations.

Relation	Arguments	Informal Definition
before-start ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(before-start occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the beginning timepoint of occ1 is earlier than the beginning time point of occ2.
before-end ----	activity-occurrence, activity-occurrence,	(before-end occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that

---	activity-occurrence	the beginning timepoint of occ1 is earlier than the ending time point of occ2.
after-start --- ---	activity-occurrence, activity-occurrence, activity-occurrence	(after-start occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the ending timepoint of occ1 is earlier than the beginning time point of occ2.
after-end --- -----	activity-occurrence, activity-occurrence, activity-occurrence	(after-end occ1 occ2 occ3) means that occ1 and occ2 are two activity occurrences that are subactivity occurrences of occ3 and that the ending timepoint of occ1 is earlier than the ending timepoint of occ2.
meets --- -----	activity-occurrence, activity-occurrence, activity-occurrence	(meets occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the ending timepoint of occ1 is equal to the beginning timepoint of occ2.
starts ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(starts occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the beginning timepoint of occ1 is equal to the beginning timepoint of occ2.
finishes --- ---	activity-occurrence, activity-occurrence, activity-occurrence	(finishes occ1 occ2 occ3) means that occ1 and occ2 are two activity occurrences that are subactivity occurrences of occ3 and that the ending timepoint of occ1 is equal to the ending timepoint of occ2.
during ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(during occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the beginning and ending timepoints of occ1 are between the beginning and ending timepoints of occ2.
overlaps ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(overlaps occ1 occ2 occ3) means that occ1 and occ2 are two activity occurrences that are subactivity occurrences of occ3 and that the beginning timepoint of occ1 is between the beginning and ending timepoints of occ2, and the ending timepoint of occ2 is between the beginning and ending timepoint of occ1.
equals ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(equals occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the two activity-occurrences have same beginning and ending timepoints.
non-concurrence ----- ----- or ----- -----	activity-occurrence, activity-occurrence, activity-occurrence	(non-concurrence occ1 occ2 occ3) means that occ1 and occ2 are two activity-occurrences that are subactivity occurrences of occ3 and that the two activity-occurrences do not overlap.
follows	activity, activity, activity	(follows a1 a2 a3) means that a1 and a2 are two activities that are subactivities of a3 and that the ending timepoint of any activity-occurrence of a1 is earlier than or equal to the beginning timepoint of any occurrence of a2.
leq-expect	activity, activity, activity	(leq-expect a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a1 occurs, a2 must occur after the activity-occurrence of a1.
next-expect-activity	activity, activity, activity	(next-expect-activity a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a1 occurs, a2 must occur after the activity-occurrence of a1, and no other activity can occur between the activity-occurrences of a1 and

		a2.
leq-required	activity, activity, activity	(leq-required a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a2 occurs, a1 must occur before the activity-occurrence of a2.
next-required-activity	activity, activity, activity	(next-required-activity a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a2 occurs, a1 must occur before a2, and no other activity can occur between the activity-occurrences of a1 and a2.
mutually-occurring	activity, activity, activity	(mutually-occurring a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a2 occurs, a1 must occur before a2, whenever a1 occurs, a2 must occur after a1, and no other activity can occur between the occurrences of a1 and a2.
start-synchronization	activity, activity, activity	(start-synchronization a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a1 and a2 occurs, their activity-occurrences must start at the same time.
end-synchronization	activity, activity, activity	(end-synchronization a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a1 and a2 occurs, their activity-occurrences must end at the same time.
full-synchronization	activity, activity, activity	(full-synchronization a1 a2 a3) means that a1, a2, and a3 are activities, that a1 and a2 are subactivities of a3 and that whenever a1 and a2 occurs, their activity-occurrences must start and end at the same times.

The formal definitions for the relations in the Complex Sequences of Activities Extension will be completed for the next release.

4.9 Nondeterministic Activities Extension

This extension introduces concepts for describing special constraints on the occurrences of activities that are related through the subactivity relation.

4.9.1 Classes of Activities in the Nondeterministic Activities Extension

The nondeterministic activities extension introduces the following kinds.

Concept	Informal Documentation
nondeterministic-choice	A nondeterministic choice is an activity that is such that whenever it occurs, at least one of its subactivities occurs as well.
xor	An xor is an activity that is such that whenever it occurs, one and only one of its subactivities occurs as well.

4.9.2 Formal Axioms in the Nondeterministic Activities Extension

Definition 1.

```
(defrelation nondeterministic-choice (?a) :=  
  (and (activity ?a)  
    (forall (?occ)  
      (=> (occurrence-of ?occ ?a)  
        (exist (?occ1 ?a1)  
          (and (occurrence-of ?occ1 ?a1)  
            (subactivity-occurrence ?occ1 ?occ)))))))
```

Definition 2.

```
(defrelation xor (?a) :=  
  (and (activity ?a)  
    (forall (?occ)  
      (=> (occurrence-of ?occ ?a)  
        (exist (?occ1 ?a1)  
          (and (occurrence-of ?occ1 ?a1)  
            (subactivity-occurrence ?occ1 ?occ)))  
        (forall (?occ2)  
          (=> (subactivity-occurrence ?occ2 ?occ)  
            (= ?occ1 ?occ2))))))))
```

4.10 Reasoning about States Extension

The purpose of the reasoning about states extension is to introduce and define relations to talk about how activity-occurrences and fluents effect one another.

4.10.1 Classes of Fluents in the Reasoning about States Extension

The reasoning about states extension defines the following classes of fluents.

Kind	Informal Definition
Achievement	An achievement is an activity whose effects achieve the preconditions for some other activity.
Repairable-fluent	A repairable fluent is a fluent such that, whenever the fluent does not hold, there exists an activity-occurrence that causes the fluent to hold again.
Nonrepairable-fluent	A nonrepairable fluent is a fluent such that, whenever the fluent does not hold, there exists no activity-occurrence that can achieve the fluent.

Reversible-fluent	A reversible fluent is a fluent such that, whenever the fluent holds, there exists an activity-occurrence that falsifies the fluent.
Irreversible-fluent	An irreversible fluent is a fluent such that, whenever the fluent holds, there exists no activity-occurrence that can falsify the fluent.

4.10.2 Relations In the Reasoning about States Extension

The reasoning about states extension defines the following relations.

Relation	Arguments	Informal Definition
Changes	activity-occurrence, fluent	This relation is used to capture the effect of an activity-occurrence on the properties of the world. An activity-occurrence changes a fluent if either the fluent held before the activity-occurrence and does not hold after it, or the fluent did not hold before the activity-occurrence but holds after it.
Possibly-changes	activity-occurrence, fluent, fluent	This relation is used to capture the effect of an activity-occurrence when that effect is conditional on some property of the world. An activity-occurrence possibly changes a fluent f2 given a fluent f1, if it changes f2 only when f1 holds before the activity-occurrence.
achieved	activity-occurrence, fluent	This relation is used to capture the fact that an activity-occurrence causes the world to have a certain property. An activity-occurrence achieves a fluent if the fluent does not hold before the activity-occurrence but holds after the activity-occurrence.
falsified	activity-occurrence, fluent	This relation is used to capture the fact that an activity-occurrence causes the world to not have a property that it had before the activity-occurrence. An activity-occurrence falsifies a fluent if the fluent holds before the occurrence but does not hold after the activity-occurrence.
precondition-fluent	activity, fluent	A fluent is a precondition for an activity if the fluent must hold before any activity-occurrence of that activity.
neg-precondition-fluent	activity, fluent	A fluent is a negative precondition for an activity if it must be the case that the fluent does not hold before any occurrence of that activity.
possible-fluent	fluent, timepoint	A fluent is possible fluent at a given timepoint if there exist an activity-occurrence that starts at the given timepoint and such that the fluent holds before the activity-occurrence.
required-fluent	fluent, timepoint	A fluent is possible fluent at a given timepoint if there exist an activity-occurrence that starts at the given timepoint and such that the fluent holds before the activity-occurrence.
fluent-interval	fluent, activity-occurrence, activity-occurrence	Two activity-occurrences occ1 and occ2 are an interval for a fluent if occ1 achieves the fluent, occ2 falsifies the fluent, and there are no activity-occurrences that occur between occ1 and occ2 and that falsifies the fluent.
neg-fluent-interval	fluent, activity-	Two activity-occurrences occ1 and occ2 are a negative interval for a fluent

	occurrence, activity-occurrence	if occ1 occurs before occ2, occ1 falsifies the fluent, occ2 achieves the fluent, and there are no activity-occurrences that occur between occ1 and occ2 and that falsifies the fluent.
temporal-fluent-interval	fluent, time point, timepoint	Informally, a temporal interval for a fluent is an interval over which the fluent holds. More formally, a fluent has a temporal interval beginning at time t1 and ending at time t2 if there exist two activity-occurrences occ1 and occ2 such that occ1 starts at t1, occ2 ends at t2 and occ1 and occ2 form an interval for the fluent.

4.10.3 Formal Definitions for the Reasoning about States Extension

Definition 1.

```
(defrelation changes (?occ ?f) :=
  (or (and (state-before ?f ?occ)
    (not (state-after ?f ?occ)))
    (and (not (state-before ?f ?occ))
      (state-after ?f ?occ))))
```

Definition 2.

```
(defrelation possibly-changes (?occ ?f1 ?f2) :=
  (and (state-before ?f2 ?occ)
    (or (and (state-before ?f1 ?occ)
      (not (state-after ?f1 ?occ)))
      (and (not (state-before ?f1 ?occ))
        (state-after ?f1 ?occ))))
```

Definition 3.

```
(defrelation achieved (?occ ?f) :=
  (and (not (state-before ?f1 ?occ))
    (state-after ?f1 ?occ)))
```

Definition 4.

```
(defrelation falsified (?occ ?f) :=
  (and (state-before ?f1 ?occ)
    (not (state-after ?f1 ?occ))))
```

Definition 5.

```
(defrelation possible-fluent(?f) :=
  (exist (?occ)
    (state-before ?f1 ?occ)))
```

Definition 6.

```
(defrelation precondition-fluent(?a ?f) :=  
  (forall (?occ)  
    (=> (occurrence-of ?occ ?a)  
        (state-before ?f ?occ))))
```

Definition 7.

```
(defrelation neg-precondition-fluent(?a ?f) :=  
  (forall (?occ)  
    (=> (occurrence-of ?occ ?a)  
        (not (state-before ?f ?occ)))))
```

Definition 8.

```
(defrelation possible-fluent(?f ?t) :=  
  (exist (?occ)  
    (and (= ?t (beginof ?occ))  
         (state-before ?f ?occ))))
```

Definition 9.

```
(defrelation required-fluent(?f ?t) :=  
  (forall (?occ)  
    (=> (and (activity-occurrence ?occ)  
            (= ?t (beginof ?occ))  
            (state-before ?f ?occ)))))
```

Definition 10.

```
(defrelation repairable-fluent(?f) :=  
  (forall (?occ)  
    (=> (not (state-after ?f ?occ))  
        (exist (?occ1)  
          (and (occurrence-earlier ?occ ?occ1)  
               (achieved ?occ1 ?f))))))
```

Definition 10.

```
(defrelation non-repairable-fluent(?f) :=  
  (forall (?occ)  
    (=> (not (state-after ?f ?occ))  
        (not (exist (?occ1)  
          (and (occurrence-earlier ?occ ?occ1)  
               (achieved ?occ1 ?f)))))))
```

Definition 11.

```
(defrelation reversible-fluent(?f) :=  
  (forall (?occ)  
    (=> (state-after ?f ?occ)  
      (exist (?occ1)  
        (and (occurrence-earlier ?occ ?occ1)  
              (falsifies ?occ1 ?f))))))
```

Definition 12.

```
(defrelation irreversible-fluent(?f) :=  
  (forall (?occ)  
    (=> (state-after ?f ?occ)  
      (not (exist (?occ1)  
        (and (occurrence-earlier ?occ ?occ1)  
              (falsifies ?occ1 ?f))))))
```

Definition 13.

```
(defrelation fluent-interval(?f ?occ1 ?occ2) :=  
  (and (occurrence-earlier ?occ1 ?occ2)  
    (achieved ?f ?occ1)  
    (falsified ?f ?occ2)  
    (forall (?occ)  
      (=> (and (not (= ?occ ?occ1))  
        (not (= ?occ ?occ2))  
        (occurrence-earlier ?occ1 ?occ)  
        (occurrence-earlier ?occ ?occ2))  
        (state-after ?occ ?f))))))
```

Definition 14.

```
(defrelation neg-fluent-interval(?f ?occ1 ?occ2) :=  
  (and (occurrence-earlier ?occ1 ?occ2)  
    (achieved ?f ?occ2)  
    (falsified ?f ?occ1)  
    (forall (?Socc)  
      (=> (and (not (= ?occ ?occ1))  
        (not (= ?occ ?occ2))  
        (occurrence-earlier ?occ1 ?occ)  
        (occurrence-earlier ?occ ?occ2))  
        (not (state-after ?occ ?f))))))
```


Definition 15.

```
(defrelation temporal-fluent-interval(?f ?t1 ?t2) :=  
  (exist (?occ1 ?occ2)  
    (and (= ?t1 (beginof ?occ1))  
          (= ?t2 (endof ?occ2))  
          (fluent-interval ?f ?occ1 ?occ2))))
```

4.11 Interval Activities Extension

The purpose of this extension is to introduce the concept of interruptible and uninterruptible activities.

4.11.1 Classes of Activities in the Interval Activities Extension

The interval activities extension introduces the following kinds.

Kind	Informal Definition
interval-activity	An interval activity is an activity that has two subactivities: one that occurs at the beginning (the initial subactivity) and one that occurs at the end of each occurrence of the activity (the terminal subactivity). The initial activity establishes a particular property of the world, namely, the property of that occurrence of the activity being initiated. The terminal activity falsifies that property. That property is called the 'activity-fluent.'
uninterruptible-activity	An uninterruptible activity is an interval activity that cannot be resumed once it has been stopped.
interruptible	An interruptible activity is an interval activity that can be resumed if it is stopped.

4.11.1.1 Interval-Activity

Interval activities are arguments to the following relations.

Relation	Arguments	Informal Definition
suspends	activity, interval-activity	An activity suspends an interval activity if its activity-occurrence causes the interval activity to be suspended.

Interval activities are arguments to the following functions.

Function	Arguments	Return value	Informal Definition
initiate	interval-activity	primitive-activity	This function returns the primitive activity that occurs when the interval activity begins to be executed.

terminate	interval-activity	primitive-activity	This function returns the primitive activity that occurs when the interval activity ends its execution.
activity-fluent	Interval-activity	state	This function returns the state which holds only between occurrences of the initiation and termination of an interval activity.
executing	Interval-activity	state	This function returns the state which holds during the occurrence of an interval activity.
suspended	interval-activity	state	This function returns the state which holds after the occurrence of a suspending activity.

4.11.2 Defined Functions in the Interval Activities Extension

The following functions are defined in the interval activities extension.

Function	Arguments	Return type	Informal Definition
initiate	interval-activity	activity	This function returns the initial activity of an interval activity
terminate	interval-activity	activity	This function returns the terminal activity of an interval activity.
executing	Interval-activity	state	This function returns the state which holds during the occurrence of an interval activity.
suspended	interval-activity	state	This function returns the state which holds after the occurrence of a suspending activity.

4.11.3 Defined Relations in the Interval Activities Extension

The following relations are defined in the interval activities extension.

Relation	Arguments	Informal Definition
suspends	activity, interval-activity	An activity suspends an interval activity if its activity-occurrence causes the interval activity to be suspended.

4.11.4 Formal Definitions and Axioms for the Interval Activities Extension

Definition 1.

```
(defrelation interval-activity (?a) :=
  (forall (?occ)
    (<=> (occurrence-of ?occ ?a))
```

```

(exists (?a1 ?a2 ?occl ?occ2)
  (and (subactivity ?a1 ?a)
        (subactivity ?a2 ?a)
        (= ?a1 (initiate ?a))
        (= ?a2 (terminate ?a))
        (subactivity-occurrence ?occl ?occ)
        (subactivity-occurrence ?occ2 ?occ)
        (= (beginof ?occl) (beginof ?occ))
        (= (endof ?occl) (endof ?occ))))))

```

Definition 2.

```

(deffunction initiate (?a) :=
  (forall (?occ)
    (=> (occurrence-of ?occ (initiate ?a))
        (holds-after (activity-fluent ?a) ?occ))))

```

Definition 3.

```

(deffunction terminate (?a) :=
  (forall (?occ)
    (=> (occurrence-of ?occ (terminate ?a))
        (not (holds-after (activity-fluent ?a) ?occ)))))

```

Definition 4.

```

(deffunction activity-fluent (?a) :=
  (forall (?ap ?occ)
    (=> (occurrence-of ?occ ?ap)
        (< => (holds-after (activity-fluent ?a) ?occ)
              (or (= ?ap (initiate ?a))
                  (and (hold-before (activity-fluent ?a) ?occ)
                        (not (= ?ap (terminate ?a))))))))))

```

Definition 5.

```

(deffunction executing (?a) :=
  (forall (?a ?occ)
    (<=> (holds-before (executing ?a) ?occ)
        (and (holds-before (activity-fluent ?a) ?occ)
              (legal-activity (terminate ?a) ?occ)))))

```

Definition 6.

```
(defunction suspended (?a) :=  
(forall (?a ?occ)  
  (<=> (holds-before (suspended ?a) ?occ)  
    (and (holds-before (activity-fluent ?a) ?occ)  
      (not (legal-activity (terminate ?a) ?occ))))))
```

Definition 7.

```
(defrelation suspends (?a1 ?a2) :=  
(forall (?occ)  
  (=> (occurrence-of ?occ ?a1)  
    (holds-after (suspended ?a2) ?occ))))
```

Definition 8.

```
(defrelation uninterruptible (?a) :=  
(nonrepairable (executing ?a)))
```

Definition 9.

```
(defrelation interruptible (?a) :=  
(repairable (executing ?a)))
```

4.12 Temporal Ordering Relations Extension

The purpose of the temporal ordering relation extension is to define relations that allow a temporal ordering of activities that include the notion of delay.

4.12.1 Defined Relations in the Temporal Ordering Extension

The temporal ordering extension defines the following relations.

Relation	Arguments	Informal Definition
before-start-delay	activity-occurrence, activity-occurrence, activity-occurrence, duration	(before-start-delay occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 begins at least d timepoints after occ1 begins.
before-end-delay	activity-occurrence, activity-occurrence, activity-occurrence, duration	(before-end-delay occ1 occ2 occ d) means that occ1 and occ2 are sub activity occurrences of occ and that occ2 starts at least d timepoints after occ1 ends.
after-start-delay	activity-occurrence, activity-occurrence,	(after-start-delay occ1 occ2 occ d) means that occ1 and occ2 are sub activity occurrences of occ and that occ2 ends at least d timepoints after occ1 begins.

	activity-occurrence, duration	
after-end-delay	activity-occurrence, activity-occurrence, activity-occurrence, duration	(after-end-delay occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 ends at least d timepoints after occ1 ends.
start-equal-start	activity-occurrence, activity-occurrence, activity-occurrence, duration	(start-equal-start occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 begins exactly d timepoints after occ1 begins.
start-equal-end	activity-occurrence, activity-occurrence, activity-occurrence, duration	(start-equal-end occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 begins exactly d timepoints after occ1 ends.
end-equal-start	activity-occurrence, activity-occurrence, activity-occurrence, duration	(end-equal-start occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 ends exactly d timepoints after occ1 begins.
end-equal-end	activity-occurrence, activity-occurrence, activity-occurrence, duration	(end-equal-end occ1 occ2 occ d) means that occ1 and occ2 are subactivity occurrences of occ and that occ2 ends exactly d timepoints after occ1 ends.

4.12.2 Formal Definitions for the Temporal Ordering Extension

Definition 1.

(defrelation before-start-delay (?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (beforeEq (timeAdd (beginof ?occ1) ?d) (beginof ?occ2))))

Definition 2.

(defrelation before-start-delay (?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (beforeEq (timeAdd (endof ?occ1) ?d) (beginof ?occ2))))

Definition 3.

(defrelation after-start-delay (?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)

(beforeEq (timeAdd (beginof ?occ1) ?d) (endof ?occ2))))

Definition 4.

(defrelation after-end-delay (?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (beforeEq (timeAdd (endof ?occ1) ?d) (endof ?occ2)))))

Definition 5.

(defrelation start-equal-start(?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (= (timeAdd (beginof ?occ1) ?d) (beginof ?occ2))))

Definition 6.

(defrelation start-equal-end(?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (= (timeAdd (endof ?occ1) ?d) (beginof ?occ2))))

Definition 7.

(defrelation end-equal-star(?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (= (timeAdd (beginof ?occ1) ?d) (endof ?occ2))))

Definition 8.

(defrelation end-equal-end(?occ1 ?occ2 ?occ ?d) :=
 (and (subactivity-occurrence ?occ1 ?occ)
 (subactivity-occurrence ?occ2 ?occ)
 (= (timeAdd (endof ?occ1) ?d) (endof ?occ2))))

4.13 Junctions Extension

The purpose of the junctions extension is to introduce the concepts of branching (decision and parallel activity-occurrences) and synchronization of activity-occurrences.

4.13.1 Classes of Activities in the Junctions Extension

The junctions extension defines the following kinds.

Kind	Informal Definition
or-split	An or-split is an activity such that whenever that activity occurs, at least one of its subactivities occur as well.
and-split	An and-split is an activity such that whenever that activity occurs, all of its subactivities occur as well.
xor-split	An xor-split is an activity such that whenever that activity occurs, one and only one of its subactivities occur as well.
sync-start	A sync-start activity is an activity such that whenever the activity occurs, if two or more of its subactivities occur, their activity-occurrences start at the same time.
sync-finish	A sync-finish activity is an activity such that whenever the activity occurs, if two or more of its subactivities occur, their activity-occurrences end at the same time.
sync-start-and-split	A sync-start-and-split activity is an activity that is both a sync-start and an and-split activity.
sync-start-or-split	A sync-start-or-split activity is an activity that is both a sync-start and an or-split activity.
sync-finish-and-split	A sync-finish-and-split activity is an activity that is both a sync-finish and an and-split activity.
sync-finish-or-split	A sync-finish-or-split activity is an activity that is both a sync-finish and an or-split activity.

4.13.2 Formal Definitions for the Junctions Extension

Definition 1.

```
(defrelation or-split (?a) :=
  (and (activity ?a)
    (forall (?occ)
      (=> (occurrence-of ?occ ?a)
        (exists (?occ1)
          (subactivity-occurrence ?occ1 ?occ))))))
```

Definition 2.

```
(defrelation and-split (?a) :=
  (and (activity ?a)
    (forall (?occ ?a1)
      (=> (and (occurrence-of ?occ ?a)
        (subactivity ?a1 ?a))
        (exists (?occ1)
          (and (occurrence-of ?occ1 ?a1)
            (subactivity-occurrence ?occ1 ?occ))))))
```

Definition 3.

```
(defrelation xor-split (?a) :=  
  (and (activity ?a)  
    (forall (?occ)  
      (=>(occurrence-of ?occ ?a)  
        (and (exists (?occ1)  
          (subactivity-occurrence ?occ1 ?occ))  
          (forall (?occ2 ?occ3)  
            (=> (and (subactivity-occurrence ?occ2 ?occ)  
              (subactivity-occurrence ?occ3 ?occ))  
              (= ?occ2 ?occ3))))))))))
```

Definition 4.

```
(defrelation sync-start(?a) :=  
  (and (activity ?a)  
    (forall (?occ ?occ1 ?occ2)  
      (=> (and (occurrence-of ?occ ?a)  
        (subactivity-occurrence ?occ1 ?occ)  
        (subactivity-occurrence ?occ2 ?occ))  
        (= (beginof ?occ1) (beginof ?occ2))))))
```

Definition 5.

```
(defrelation sync-finish(?a) :=  
  (and (activity ?a)  
    (forall (?occ ?occ1 ?occ2)  
      (=> (and (occurrence-of ?occ ?a)  
        (subactivity-occurrence ?occ1 ?occ)  
        (subactivity-occurrence ?occ2 ?occ))  
        (= (endof ?occ1) (endof ?occ2))))))
```

Definition 6.

```
(defrelation sync-start-and-split(?a) :=  
  (and (sync-start ?a)  
    (and-split ?a)))
```

Definition 7.

```
(defrelation sync-start-or-split(?a) :=  
  (and (sync-start ?a)  
    (or-split ?a)))
```


Definition 8.

```
(defrelation sync-finish-and-split(?a) :=  
  (and (sync-finish ?a)  
        (and-split ?a)))
```

Definition 9.

```
(defrelation sync-finish-or-split(?a) :=  
  (and (sync-finish ?a)  
        (or-split ?a)))
```

5 Translation Using PSL

5.1 Motivation

To guarantee correct and complete translation, translators must be based on the *formal* specifications of the representation's semantics. Translators written "by hand" provide no such guarantee, and proving that they actually perform the intended "correct" translation is so difficult that it is almost never done.

5.2 Overview of Semantic and Syntactic Translation

We consider translation to be a two-stage process -- syntactic translation and semantic translation. The syntactic translator is a parser between the PSL syntax (e.g. KIF) and the native syntax of one of the applications; this parser keeps the terminology of the application intact. Figure 5 illustrates the translation transaction between two applications and the role played by the PSL Ontology.

Semantic translation substitutes terminology of one application with PSL definitions. These translation definitions between an application ontology and PSL can be derived from the ontological definitions that were written using the same foundational theories. These are definitions for the terminology of the application ontology, using *only* the terminology from the PSL Ontology, as well as definitions for the terminology of the PSL Ontology using *only* the terminology of the application ontology.

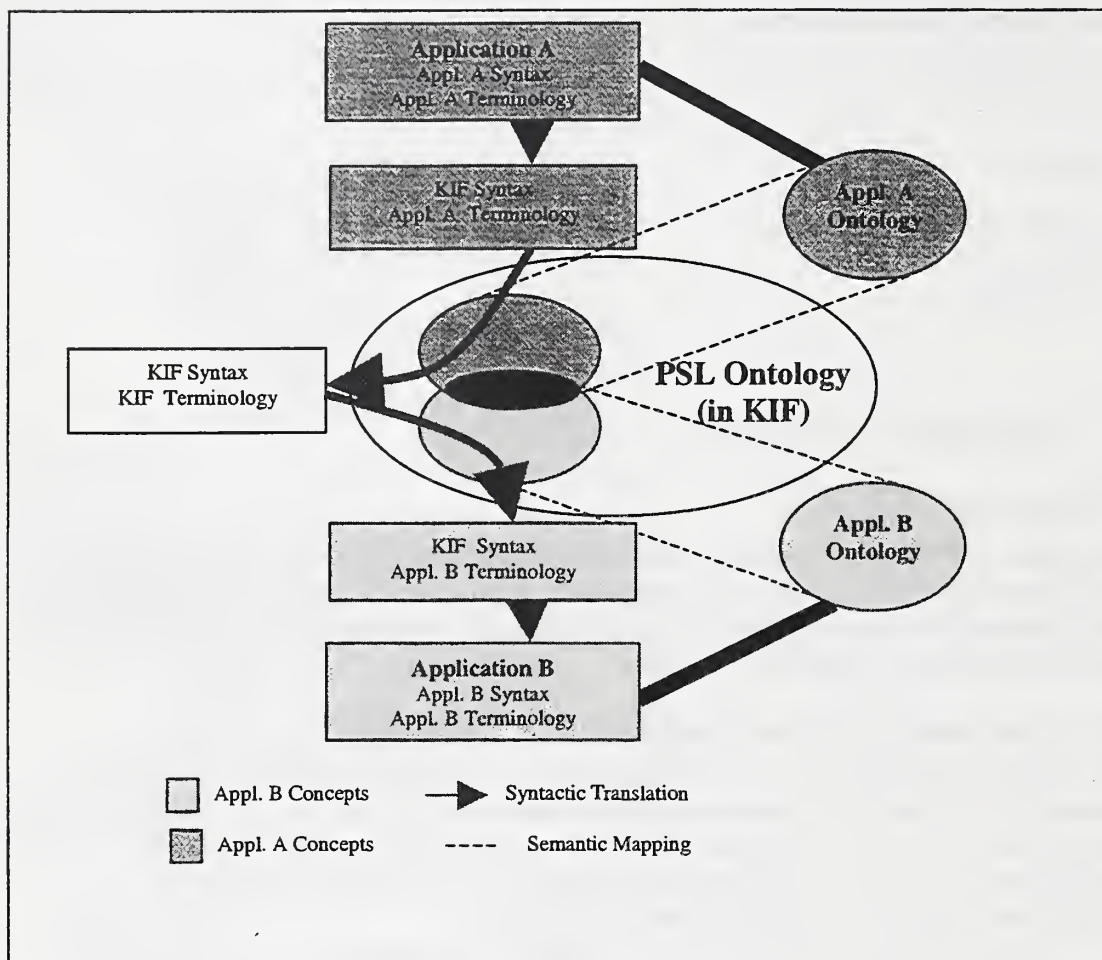


Figure 5: Translation to/from PSL

This procedure is best shown by an example. The resource construct is highlighted during each stage of the example to show how it progresses through the translation process. We begin with a simple file written in Application A's syntax and using Application A's terminology.

```
{stock: wire (x)}
{stock: plug (x)}
{resource: inject_mold (x)}
{material: plug_head (x)}
{operation: fabricate_plug}
```

The syntactic translator takes this file and produces a corresponding file using PSL syntax, but still preserving Application A terminology.

```
(forall (?r)
  (=> (wire ?r)
      (stock ?r)))
```

```
(forall (?r)
  (=> (plug_head ?r)
      (material ?r)))
```

```
(forall (?r)
  (=> (inject_mold ?r)
      (resource ?r)))
```

```
(forall (?r)
  (=> (plug ?r)
      (stock ?r)))
```

The semantic translator takes this file and produces a file containing only PSL terminology by substituting the definitions of all Application A terms with their definitions in PSL.

```
(forall (?r)
  (=> (wire ?r)
      (material ?r)))
```

```
(forall (?r)
  (=> (plug_head ?r)
      (wip ?r)))
```

```
(forall (?r)
  (=> (inject_mold ?r)
      (machine ?r)))
```

```
(forall (?r)
  (=> (plug ?r)
      (material ?r)))
```

We now follow reversed steps to translate the file into Application B. Using the translation definitions for Application B, the PSL file is mapped to a file containing only Application B terminology.

```
forall (?r)
  (=> (wire ?r)
      (resource ?r)))
```

```
(forall (?r)
```

```

      (=> (plug_head ?r)
          (workpiece ?r)))

  (forall (?r)
    (=> (inject_mold ?r)
        (machine-tool ?r)))

  (forall (?r)
    (=> (plug ?r)
        (resource ?r)))

```

Finally, the syntactic translator for Application B maps the file back into Application B syntax.

```

(define-class wire
  (Subclass-Of resource))

(define-class inject-mold
  (Subclass-Of machine-tool))

(define-class plug
  (Subclass-Of resource))

(define-class plug_head
  (Subclass-Of workpiece))

(define-class fabricate_plug
  (Subclass-Of task))

```

Note that this ontology-based approach to compliance is different from the traditional approach to standards compliance. Rather than forcing the adoption of exactly the same terminology, an application is PSL-compliant if there exist definitions for its terminology using either some foundational theory or other ontology. Given these definitions, translation definitions can be written between the application and PSL.

6 Conclusion

The purpose of the Process Specification Language is to provide a representation for manufacturing process information that will serve as an Interlingua to facilitate the exchange of information among manufacturing software applications. This paper documents Version 1.0 of PSL.

Other efforts to develop mechanisms for the exchange of data, such as ISO 10303 (informally known as The STandard for the Exchange of Product model data (STEP)) [16], have focused on syntactical standards elements necessary for data exchange. This focus works well for exchanging information among similar domains where the terms used have roughly the same meanings. However, within the increasingly complex manufacturing environment where process models are maintained in different software

applications, standards for the exchange of this information must address not only the syntax but also the meanings or semantics of terms and concepts used. PSL uniquely addresses this in its identification and development of semantics for specifying and exchanging process information. The identification of the necessary concepts was based on a thorough analysis of the requirements for specifying business and manufacturing engineering processes in the manufacturing domain and then analyzing a broad set of existing approaches to representing process models with respect to these requirements.

Version 1.0 of PSL represents the beginning point in the development of a complete Process Specification Language. This initial version will be refined in an iterative fashion to continuously increase the robustness of the language. A series of pilot implementations, in which the specification language will be used to exchange process information between existing manufacturing applications, will allow us to determine which representational areas need to be expanded upon to ensure that the PSL will be able to capture and exchange all current and future manufacturing process information.

7 References

- [1] Schlenoff, C., Knutilla, A., Ray, S., Unified Process Specification Language: Requirements for Modeling Processes: NISTIR 5910, 1996, National Institute of Standards and Technology, Gaithersburg, MD.
- [2] Knutilla, A., et al., Process Specification Language: An Analysis of Existing Representations, NISTIR 6133, 1998, National Institute of Standards and Technology, Gaithersburg, MD.
- [3] Genesereth, M., Fikes, R.: Knowledge Interchange Format (Version 3.0) - Reference Manual, 1992, Computer Science Dept., Stanford University, Stanford, CA.
- [4] Catron, B., Ray, S., ALPS: A Language for Process Specification, Int. J. Computer Integrated Manufacturing, 1991, Vol. 4, No. 2, 105-113.
- [5] Fox, M., et al, An Organization Ontology for Enterprise Modeling: Preliminary Concepts," Computers in Industry, 1996, Vol. 19, pp. 123-134.
- [6] Uschold, M., et. al., "The Enterprise Ontology," The Knowledge Engineering Review, Vol. 13(1), pp. 31-89, Special Issue on "Putting Ontologies to Use," (eds. Uschold, M. and Tate, A.), Cambridge University Press.
- [7] Pease, A., Core Plan Representation (CPR), <http://www.teknowledge.com/CPR2/>, November 13, 1998.
- [8] Tate, A., "Roots of SPAR – Shared Planning and Activity Representation," The Knowledge Engineering Review, Vol. 13(1), pp. 121-128, Special Issue on "Putting Ontologies to Use" (eds. Uschold, M. and Tate, A.), Cambridge University Press.
<http://www.aiai.ed.ac.uk/~arpi/spar/>
- [9] Lee, J., et al, "The PIF Process Interchange Format and Framework," The Knowledge Engineering Review, Vol. 13(1), pp. 91-120, Special Issue on "Putting Ontologies to Use" (eds. Uschold, M. and Tate, A.), Cambridge University Press.
- [10] Workflow Management Coalition Members, Glossary: A Workflow Management Coalition Specification, Belgium, 1994.
- [11] Uschold, M. and Gruninger M., Ontologies: Principles, Methods, and Applications, Knowledge Engineering Review, 1996, Vol. 11, pp. 96-137.
- [12] Reiter, R., The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression, Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, 1991, pages 418-440, Academic Press, San Diego.

[13] PSL Ontology – Current Theories and Extensions, <http://www.nist.gov/psl/psl-ontology/>, June 28, 1999.

[14] Hayes, P., A Catalog of Temporal Theories, Tech Report UIUC-BI-AI-96-01, 1996, University of Illinois.

[15] Smith, S.J.J., et al., Integrating Electrical and Mechanical Design and Process Planning, Proceedings of the IFIP Knowledge Intensive CAD Workshop, 1996, Carnegie-Mellon University (CMU).

[16] ISO 10303-1:1994, Product data representation and exchange: Part 1: Overview and fundamental principles.

[17] Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998, <http://www.w3.org/TR/1998/REC-xml-19980210>.

[18] Resource Description Framework (RDF) Model and Syntax, W3C Recommendation 22 February 1999, <http://www.w3.org/TR/REC-rdf-syntax/>.

[19] Resource Description Framework (RDF) Schema Specification, W3C Proposed Recommendation 03 March 1999, <http://www.w3.org/TR/PR-rdf-schema/>.

[20] ISO 10303-11: 1994, Product data representation and exchange: Part 11: EXPRESS language reference manual

Appendix A: Sample PSL Instance

The following is an example instance of a PSL exchange file. For the completed pilot implementation, the project has used a KIF-like syntax for the exchange (as shown below). Future versions of this PSL specification document will go into more detail about the syntax and grammar of the PSL exchange language, which may or may not resemble KIF, as the specification evolves.

```
(doc make-gt350 "Make GT350")
```

```
(and (doc make-interior "Make Interior")
      (forall (?a : (activation-of ?a make-interior))
        (exists (?o1 : (instance-of ?o1 a002-bench)) (in ?o1 ?a)))
      (and (duration make-interior 5) (beginof make-interior 7)))
```

```
(and (doc make-drive "Make Drive")
      (forall (?a : (activation-of ?a make-drive))
        (exists (?o1 : (instance-of ?o1 a003-bench)) (in ?o1 ?a))))
```

```
(and (doc make-trim "Make Trim")
      (forall (?a : (activation-of ?a make-trim))
        (exists
          (?o1 ?o2 : (instance-of ?o1 a002-bench)
                    (instance-of ?o2 a005-bench))
          (and (in ?o1 ?a) (in ?o2 ?a)))))
```

```
(doc make-chassis "Make Chassis")
```

```
(doc final-assembly "Final Assembly")
```

```
(doc make-harness "Make Harness")
```

```
(doc make-wires "Make Wires")
```

```
(doc assemble-engine "Assemble Engine")
```


(doc machine-block "Machine Block")

(doc change-mould "Change Mould")

(doc setup-furnace "Setup Furnace")

(doc analyze-metal "Analyze Metal")

(doc melt "Melt")

(doc wait "Wait")

(doc clear-cavities "Clear Cavities")

(doc setup-racks "Setup Racks")

(doc pour "Pour")

(doc remove-racks "Remove Racks")

(doc batch-complete "Batch Complete")

(and (doc make-gt350-proc "Make GT350 Process")

 (subactivity make-interior-1 make-gt350-proc)

 (subactivity make-drive-1 make-gt350-proc)

 (subactivity make-trim-1 make-gt350-proc)

 (subactivity make-chassis-1 make-gt350-proc)

 (subactivity final-assembly-1 make-gt350-proc)

 (subactivity j2 make-gt350-proc) (subactivity j1 make-gt350-proc)

 (subactivity make-harness-1 make-gt350-proc)

 (subactivity make-wires-1 make-gt350-proc)

 (subactivity assemble-engine-1 make-gt350-proc)

 (subactivity j4 make-gt350-proc) (subactivity j3 make-gt350-proc)

 (subactivity machine-block-1 make-gt350-proc)

```

(subactivity change-mould-1 make-gt350-proc)
(subactivity setup-furnace-1 make-gt350-proc)
(subactivity analyse-metal-1 make-gt350-proc)
(subactivity melt-1 make-gt350-proc)
(subactivity wait-1 make-gt350-proc)
(subactivity clear-cavities-1 make-gt350-proc)
(subactivity j8 make-gt350-proc) (subactivity j7 make-gt350-proc)
(subactivity j6 make-gt350-proc) (subactivity j5 make-gt350-proc)
(subactivity setup-racks-1 make-gt350-proc)
(subactivity pour-1 make-gt350-proc)
(subactivity remove-racks-1 make-gt350-proc)
(subactivity batch-complete-1 make-gt350-proc)
(idef-process make-gt350-proc))

```

```

(and (doc make-interior-1
      "The occurrence of Make-Interior in the Dec-19 schematic")
      (forall (?a : (activation-of ?a make-interior-1))
        (activation-of ?a make-interior))
      (forall (?a : (activation-of ?a make-interior-1))
        (exists (?p : (activation-of ?p make-gt350-proc))
          (subactivity-occurrence ?a ?p))))

```

```

(and (doc make-drive-1
      "The occurrence of Make-Drive in the Dec-19 schematic")
      (forall (?a : (activation-of ?a make-drive-1))
        (activation-of ?a make-drive))
      (forall (?a : (activation-of ?a make-drive-1))
        (exists (?p : (activation-of ?p make-gt350-proc))
          (subactivity-occurrence ?a ?p))))

```

```

(and (doc make-trim-1
      "The occurrence of Make-Trim in the Dec-19 schematic")
      (forall (?a : (activation-of ?a make-trim-1))
        (activation-of ?a make-trim))
      (forall (?a : (activation-of ?a make-trim-1))
        (exists (?p : (activation-of ?p make-gt350-proc))
          (subactivity-occurrence ?a ?p))))

```

```

(and (doc make-chassis-1
      "The occurrence of Make-Chassis in the Dec-19 schematic")
      (forall (?a : (activation-of ?a make-chassis-1))
        (activation-of ?a make-chassis))

```

```
(forall (?a : (activation-of ?a make-chassis-1))
(exists (?p : (activation-of ?p make-gt350-proc))
(subactivity-occurrence ?a ?p))))
```

```
(and (doc final-assembly-1
"The occurrence of Final-Assembly in the Dec-19 schematic")
(forall (?a : (activation-of ?a final-assembly-1))
(activation-of ?a final-assembly))
(forall (?a : (activation-of ?a final-assembly-1))
(exists (?p : (activation-of ?p make-gt350-proc))
(subactivity-occurrence ?a ?p))))
```

```
(and (doc j2 "J2")
(forall (?j : (activation-of ?j j2))
(exists (?p : (activation-of ?p make-gt350-proc))
(subactivity-occurrence ?j ?p)))
(follows j2 final-assembly-1 make-gt350-proc)
(and (and_split j2 make-gt350-proc)
(subactivity make-interior-1 j2)
(subactivity make-drive-1 j2) (subactivity make-trim-1 j2)
(subactivity assemble-engine-1 j2)
(subactivity make-chassis-1 j2)))
```

```
(and (doc j1 "J1")
(forall (?j : (activation-of ?j j1))
(exists (?p : (activation-of ?p make-gt350-proc))
(subactivity-occurrence ?j ?p)))
(and (and_split j1 make-gt350-proc)
(subactivity make-interior-1 j1)
(subactivity make-drive-1 j1) (subactivity make-trim-1 j1)
(subactivity j3 j1) (subactivity make-chassis-1 j1)))
```

```
(and (doc make-harness-1
"The occurrence of Make-Harness in the Dec-26 schematic")
(forall (?a : (activation-of ?a make-harness-1))
(activation-of ?a make-harness))
(forall (?a : (activation-of ?a make-harness-1))
(exists (?p : (activation-of ?p make-gt350-proc))
(subactivity-occurrence ?a ?p))))
```

```
(and (doc make-wires-1
```

```

    "The occurrence of Make-Wires in the Dec-26 schematic")
  (forall (?a : (activation-of ?a make-wires-1))
    (activation-of ?a make-wires))
  (forall (?a : (activation-of ?a make-wires-1))
    (exists (?p : (activation-of ?p make-gt350-proc))
      (subactivity-occurrence ?a ?p))))

  (and (doc assemble-engine-1
    "The occurrence of Assemble-Engine in the Dec-26 schematic")
    (forall (?a : (activation-of ?a assemble-engine-1))
      (activation-of ?a assemble-engine))
    (forall (?a : (activation-of ?a assemble-engine-1))
      (exists (?p : (activation-of ?p make-gt350-proc))
        (subactivity-occurrence ?a ?p))))

  (and (doc j4 "J4")
    (forall (?j : (activation-of ?j j4))
      (exists (?p : (activation-of ?p make-gt350-proc))
        (subactivity-occurrence ?j ?p)))
    (follows j4 assemble-engine-1 make-gt350-proc)
    (and (and_split j4 make-gt350-proc)
      (subactivity machine-block-1 j4)
      (subactivity make-harness-1 j4)
      (subactivity make-wires-1 j4)))

  (and (doc j3 "J3")
    (forall (?j : (activation-of ?j j3))
      (exists (?p : (activation-of ?p make-gt350-proc))
        (subactivity-occurrence ?j ?p)))
    (and (and_split j3 make-gt350-proc) (subactivity j5 j3)
      (subactivity make-harness-1 j3)
      (subactivity make-wires-1 j3)))

  (and (doc machine-block-1
    "The occurrence of Machine-Block in the Dec-27 schematic")
    (forall (?a : (activation-of ?a machine-block-1))
      (activation-of ?a machine-block))
    (forall (?a : (activation-of ?a machine-block-1))
      (exists (?p : (activation-of ?p make-gt350-proc))
        (subactivity-occurrence ?a ?p))))

```


(and (doc 193 "L93")
 (follows clear-cavities-1 machine-block-1 make-gt350-proc))

(and (doc change-mould-1
 "The occurrence of Change-Mould in the Dec-1 schematic")
 (forall (?a : (activation-of ?a change-mould-1))
 (activation-of ?a change-mould))
 (forall (?a : (activation-of ?a change-mould-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc setup-furnace-1
 "The occurrence of Setup-Furnace in the Dec-1 schematic")
 (forall (?a : (activation-of ?a setup-furnace-1))
 (activation-of ?a setup-furnace))
 (forall (?a : (activation-of ?a setup-furnace-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc analyse-metal-1
 "The occurrence of Analyze-Metal in the Dec-1 schematic")
 (forall (?a : (activation-of ?a analyse-metal-1))
 (activation-of ?a analyze-metal))
 (forall (?a : (activation-of ?a analyse-metal-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc melt-1 "The occurrence of Melt in the Dec-1 schematic")
 (forall (?a : (activation-of ?a melt-1)) (activation-of ?a melt))
 (forall (?a : (activation-of ?a melt-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc wait-1 "The occurrence of Wait in the Dec-1 schematic")
 (forall (?a : (activation-of ?a wait-1)) (activation-of ?a wait))
 (forall (?a : (activation-of ?a wait-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc clear-cavities-1

```

"The occurrence of Clear-Cavities in the Dec-1 schematic")
(forall (?a : (activation-of ?a clear-cavities-1))
  (activation-of ?a clear-cavities))
(forall (?a : (activation-of ?a clear-cavities-1))
  (exists (?p : (activation-of ?p make-gt350-proc))
    (subactivity-occurrence ?a ?p))))

(and (doc j8 "J8")
  (forall (?j : (activation-of ?j j8))
    (exists (?p : (activation-of ?p make-gt350-proc))
      (subactivity-occurrence ?j ?p)))
  (follows j8 setup-racks-1 make-gt350-proc)
  (and (and_split j8 make-gt350-proc)
    (subactivity analyse-metal-1 j8) (subactivity melt-1 j8)))

(and (doc j7 "J7")
  (forall (?j : (activation-of ?j j7))
    (exists (?p : (activation-of ?p make-gt350-proc))
      (subactivity-occurrence ?j ?p)))
  (follows setup-furnace-1 j7 make-gt350-proc)
  (and (and_split j7 make-gt350-proc)
    (subactivity analyse-metal-1 j7) (subactivity melt-1 j7)))

(and (doc j6 "J6")
  (forall (?j : (activation-of ?j j6))
    (exists (?p : (activation-of ?p make-gt350-proc))
      (subactivity-occurrence ?j ?p)))
  (follows j6 setup-furnace-1 make-gt350-proc)
  (and (xor_split j6 make-gt350-proc)
    (subactivity change-mould-1 j6)))

(and (doc j5 "J5")
  (forall (?j : (activation-of ?j j5))
    (exists (?p : (activation-of ?p make-gt350-proc))
      (subactivity-occurrence ?j ?p)))
  (and (xor_split j5 make-gt350-proc) (subactivity j6 j5)
    (subactivity change-mould-1 j5)))

(and (doc l121 "L121")
  (follows remove-racks-1 batch-complete-1 make-gt350-proc))

```

(and (doc l122 "L122") (follows pour-1 remove-racks-1 make-gt350-proc))

(and (doc l124 "L124") (follows setup-racks-1 pour-1 make-gt350-proc))

(and (doc l120 "L120")
 (follows batch-complete-1 wait-1 make-gt350-proc))

(and (doc l123 "L123")
 (follows wait-1 clear-cavities-1 make-gt350-proc))

(and (doc setup-racks-1
 "The occurrence of Setup-Racks in the Dec-1-1 schematic")
 (forall (?a : (activation-of ?a setup-racks-1))
 (activation-of ?a setup-racks))
 (forall (?a : (activation-of ?a setup-racks-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc pour-1 "The occurrence of Pour in the Dec-1-1 schematic")
 (forall (?a : (activation-of ?a pour-1)) (activation-of ?a pour))
 (forall (?a : (activation-of ?a pour-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc remove-racks-1
 "The occurrence of Remove-Racks in the Dec-1-1 schematic")
 (forall (?a : (activation-of ?a remove-racks-1))
 (activation-of ?a remove-racks))
 (forall (?a : (activation-of ?a remove-racks-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

(and (doc batch-complete-1
 "The occurrence of Batch-Complete in the Dec-1-1 schematic")
 (forall (?a : (activation-of ?a batch-complete-1))
 (activation-of ?a batch-complete))
 (forall (?a : (activation-of ?a batch-complete-1))
 (exists (?p : (activation-of ?p make-gt350-proc))
 (subactivity-occurrence ?a ?p))))

Appendix B: Mapping PSL Concepts to the EXPRESS Representation

Mapping the PSL Ontology to EXPRESS

Within the architecture, there are five basic concepts; all the concepts in the EXPRESS [20] presentation of PSL will fall into one (or possibly more, in the case of multiple inheritance) of these categories.

The entity *activity* represents the concept of a PSL activity specification. This can be thought of as a template in terms of possible occurrences of subactivities, described below. Very little information is associated directly with an activity; in fact, other than a human readable name, the only important aspect of an activity is its identity. All other information pertaining to the activity is captured either in the occurrences within its context, or via relations.

```
ENTITY activity;  
    name : STRING;  
END_ENTITY;
```

The entity *occurrence* represents the concept of a PSL activity occurrence. This can be an occurrence anywhere in the range from purely abstract (for example, an occurrence that may happen at some unspecified time in either the past, present, or future) to a concrete occurrence that represents something that happened or will happen at a specific time and place.

All occurrences are “occurrences of” an activity specification, and take place within the context of another activity; in other words, occurrences are “occurrences of” some subactivity of another high-level activity. Occurrences by default carry no other information (other than a human readable name). Information such as the ordering of occurrences of common subactivities are captured via relations.

```
ENTITY occurrence;  
    name : STRING;  
    occurrence_of : activity;  
    context : activity;  
END_ENTITY;
```

The entity *relation* represents an abstract supertype of many of the concepts defined within the PSL ontology, all of which relate two or more other concepts (often activities or occurrences, but sometimes other relations also). For example, a relation would be used to capture the fact that two particular (occurrences of) subactivities must occur in a certain order.

The entity *object* represents the PSL concept of object, which may be anything that is not an activity, occurrence, relation, or data value (described below). Objects can represent

physical objects, such as an NC milling machine, or people, or even conceptual objects such as a fact about a certain situation. Extensions to PSL can refine the base concept of object through subtyping.

```
ENTITY object;  
    name : STRING;  
END_ENTITY;
```

The fifth and final basic concept is that of a *data value*. This concept is used to represent miscellaneous, usually numeric types of data, for example time points and durations. This concept does not exist independently, but rather will always occur in an attribute of some other concept, typically either a relation or a more refined subtype of an activity, occurrence, or object concept. Formally, the various concepts of data values are not related in the EXPRESS model, although it is convenient to think of them in this way.

```
TYPE timepoint = SELECT (finite_value, infinite_value);  
END_TYPE;
```

These five basic concept types form the core of the EXPRESS presentation of PSL, and are grouped together into a single schema. Other EXPRESS schemas can define, in a manner mirroring the definition of an extension to the PSL ontology, extensions to the basic EXPRESS model. For example, the PSL extension dealing with ordering relations for complex sequence actions would be mapped into a new schema and would define various subtypes of the entity *relation* denoting various types of ordering relations that might hold between subactivity occurrences.

Through appropriate extensions, it is possible to create as rich a model as is desired. The basic model is quite abstract and high-level, but one can conceive of a model for specifying processes that take place on a particular NC machine tool, in which subtypes of activity, occurrence, etc. that were quite a bit more specific were used. This model would consist of the core model plus appropriate extensions to add the required semantics.

This modular approach to modeling is attractive for two reasons. First, it reduces the amount of effort necessary to develop the models in the first place, by allowing a reduction in the scope of individual extensions. Since we can always add new extensions if additional semantics are required, we can avoid the urge to model every possible concept that might conceivable occur. Secondly, from an application standpoint it is necessary only to process the extensions that a particular application understands. In particular the models are designed allow two applications to exchange data using models based on multiple extensions, only some of which they may have in common. Data defined in extensions, which fall outside of the scope of a given application, can be easily ignored.

Use of EXPRESS-X

The goal of the EXPRESS-X language is to define mappings between information models defined in EXPRESS. The EXPRESS-X language allows one to create alternate representations of EXPRESS models and mappings between EXPRESS models and other applications. These alternate representations are called views of the original models. The algorithm for deriving the entity types in a view from the entities in an original EXPRESS model is specified using various types of mapping declarations in the EXPRESS-X language.

Once an extension gets beyond the primitive semantic concepts and into more complex notions, it becomes necessary to use EXPRESS-X. The following is an example of the type of concept that is captured in EXPRESS-X.

The concept of “processor activity” is described informally in the PSL ontology as:

...an activity that uses some set of resources, consumes or modifies some other set of resources, and produces or modifies a set of objects.

The concepts of activity and use, consumption, modification, etc. of resources are described elsewhere in the ontology and are captured in EXPRESS using static modeling techniques. (In particular, they would all be subtypes of a relation between an activity and an object, the resource.) EXPRESS-X is then used to formally capture the interrelationships among those static EXPRESS structures that are implied by the informal definition above (and by the formal semantics in the ontology).

```
VIEW processor_activity;  
FROM (a : activity; r1, r2, r3 : requires)  
IDENTIFIED_BY a;  
WHERE r1.act :=: a;  
      r2.act :=: a;  
      r3.act :=: a;  
      ('RESOURCE_ROLE.REUSABLE' IN TYPEOF(r1))  
      OR ('RESOURCE_ROLE.POSSIBLY_REUSABLE' IN TYPEOF(r1));  
      ('RESOURCE_ROLE.CONSUMABLE' IN TYPEOF(r2))  
      OR ('RESOURCE_ROLE.POSSIBLY_CONSUMABLE' IN TYPEOF(r2));  
      ('PROCESSOR_ACTIONS.MODIFIES' IN TYPEOF(r3))  
      OR ('RESOURCE_QUANTITY.CREATES' IN TYPEOF(r3));  
SELECT a;  
END_VIEW;
```

The view states that a processor activity consists of a relationship between an activity instance and three distinct “requires” relationships. The unique identity of the processor activity is derived from that of the activity. Finally, the three “requires” relations are all constrained to be of certain types: the first must be reusable or possibly reusable, the

second must be (possibly) consumable, and the third must either modify or create its associated object.

One of the most powerful features of EXPRESS-X is the ability to incrementally build views on top of other views. For example, in PSL a “processor resource” is defined as an object that is a required resource of a processor activity. Using the definition of processor activity above, we can define:

```
VIEW processor_resource;  
FROM (a : processor_activity; r : requires)  
IDENTIFIED_BY r.res;  
WHERE r.act := activity;  
      ('RESOURCE_ROLE.REUSABLE' IN TYPEOF(r))  
      OR ('RESOURCE_ROLE.POSSIBLY_REUSABLE' IN TYPEOF(r));  
SELECT r.res;  
END_VIEW;
```

The use of the previously defined view for processor activity allows us to treat this as a primitive concept, like activity or object, even though it is really a complex interrelationship among other concepts.

Appendix C: Mapping PSL Concepts to the eXtensible Markup Language (XML) Representation

XML's Strengths and Weaknesses as a Presentation Language for PSL

Vendors of mainstream software applications such as Internet browsers, database environments, and business productivity tools are either already supporting or intend to support XML in their products. Mapping PSL instances to XML will enable process specifications to be interpreted by these generic applications, lowering the barriers to data sharing.

Another advantage of XML for representing process characterizations is its "tag-centric" syntax. XML is a natural fit for representing ordered sequences and hierarchies. Thus it is well-suited for describing PSL's ordering and subactivity relationships.

The Resource Description Framework (RDF), a standard for specifying metadata, adds to XML's benefits. RDF has an XML serialization syntax, making it easy to embed resource descriptions in an XML document. Therefore, resources in an XML representation of a PSL instance can be referred to using RDF. Further, RDF Schema, a type system defined for RDF, is useful for describing PSL objects.

Although XML has many strengths as presentation format for PSL, it has a major weakness. XML is not as rich a representation as KIF, or EXPRESS for that matter. In particular, there is no straightforward way in XML to describe arbitrary constraints between data elements in an information model. Such constraints could be represented using defrelation in KIF or by means of WHERE rules in EXPRESS. Because XML is deficient when it comes to specifying constraints, its presentational abilities for PSL are limited. Exactly what those limitations are is a topic for future research, but intuitively it seems that the aspects of the PSL ontology specified in KIF with defrelation must either be implicitly represented (as can be done with ordering and subactivity relationships) or omitted in an XML presentation. Also, since portions of the ontology are hard to specify in XML, XML is not suitable as an authoring environment for PSL.

Guidelines for Mapping PSL to XML

To leverage XML's strengths while minimizing its weaknesses, we suggest some guidelines for mapping PSL to XML. To illustrate these guidelines, we use as an example a simple scenario adapted from the Camile Motor Works scenario consisting of an activity "Finish product." "Finish product" involves a "Paint activity, followed by a Sand activity, followed by another Paint" activity, and concluding with a final "Sand" activity. "Paint" has three sub-activities: "Mix paint," "Apply paint," and "Clean brush." Sanding is performed the first time using 100 grit sand paper and the second time using 200 grit sand paper.

1. Use RDF Schema to represent the objects used in a process.

Assuming that "c" is an XML namespace for Camile, the paint, brush, mixer, thinner, and sand paper in our scenario could be specified as follows:

```
<rdf:RDF>
  <Class ID="Paint"/>
  <Class ID="PaintBrush"/>
  <Class ID="PaintMixer"/>
  <Class ID="PaintThinner"/>
  <Class ID="SandPaper"/>
  <Property ID="grit">
    <rdfs:range rdf:resource="#Grit"/>
    <rdfs:domain rdf:resource="#SandPaper"/>
  </Property>
  <Class ID="Grit"/>
  <c:Grit rdf:ID="100"/>
  <c:Grit rdf:ID="200"/>
  <c:Paint rdf:ID="paint-primer"/>
  <c:Paint rdf:ID="paint-blue"/>
  <c:PaintBrush rdf:ID="brush"/>
  <c:PaintMixer rdf:ID="mixer"/>
  <c:PaintThinner rdf:ID="thinner"/>
  <c:SandPaper rdf:ID="s1">
    <c:grit rdf:resource="#100"/>
  </c:SandPaper>
  <c:SandPaper rdf:ID="s2">
    <c:grit rdf:resource="#200"/>
  </c:SandPaper>
</rdf:RDF>
```

2. Represent timepoints as sequentially ordered groups of elements, with each timepoint element having a unique identifier. If the XML application uses a Document Type Definition (DTD), the unique identifier should be represented using an ID attribute so that references to the timepoint can be made using IDREF. Each timepoint element may optionally contain character data documenting the meaning of the timepoint.

Here is what the timepoints for our scenario might look like:

```
<timepoints>
  <timepoint id="p1">start</timepoint>
  <timepoint id="p2">done mixing paint</timepoint>
  <timepoint id="p3">done applying first coat of paint</timepoint>
  <timepoint id="p4">done cleaning brush</timepoint>
  <timepoint id="p5">done sanding with 100 grit sand paper</timepoint>
  <timepoint id="p6">done mixing paint</timepoint>
  <timepoint id="p7">done applying second coat of paint</timepoint>
  <timepoint id="p8">done cleaning brush</timepoint>
```

```

    <timepoint id="p9">done sanding with 200 grit sand paper</timepoint>
  </timepoints>

```

3. For each activity, specify a unique identifier (with an ID attribute if using a DTD) and an activity name. If the activity contains subactivities, specify these within a container element. If the activity has no subactivities, specify the resources used with references to the appropriate class defined in the RDF Schema.

The "Paint" activity from our scenario could be represented as follows:

```

<activity id="a2">
  <name>Paint</name>
  <subactivities>
    <activity id="a3">
      <name>Mix paint</name>
      <requires>
        <resource rdf:resource="#Paint"/>
        <resource rdf:resource="#PaintMixer"/>
      </requires>
    </activity>
    <activity id="a4">
      <name>Apply paint</name>
      <requires>
        <resource rdf:resource="#Paint"/>
        <resource rdf:resource="#PaintBrush"/>
      </requires>
    </activity>
    <activity id="a5">
      <name>Clean brush</name>
      <requires>
        <resource rdf:resource="#PaintBrush"/>
        <resource rdf:resource="#PaintThinner"/>
      </requires>
    </activity>
  </subactivities>
</activity>

```

4. Specify occurrences of activities in sequential order with sub-activities enclosed inside parent activities. Each activity occurrence should have a beginning and ending time point and, if it cannot be decomposed into sub-activities, a list of RDF-defined resource instances it uses. References to timepoints and activities should correspond to their respective unique identifiers (and should be IDREFs if using a DTD).

The XML representing the first occurrence of the "Paint" activity from our scenario might look like this:

```

<occurrence activity="a2" begin="p1" end="p4">
  <suboccurrences>
    <!-- mix first coat -->
    <occurrence activity="a3" begin="p1" end="p2">
      <objects>
        <resource rdf:resource="#paint-primer"/>
        <resource rdf:resource="#mixer"/>
      </objects>
    </occurrence>
    <!-- apply first coat -->
    <occurrence activity="a4" begin="p2" end="p3">
      <objects>
        <resource rdf:resource="#paint-primer"/>
        <resource rdf:resource="#brush"/>
      </objects>
    </occurrence>
    <!-- clean brush -->
    <occurrence activity="a5" begin="p3" end="p4">
      <objects>
        <resource rdf:resource="#brush"/>
        <resource rdf:resource="#thinner"/>
      </objects>
    </occurrence>
  </suboccurrences>
</occurrence>

```

5. Primitive lexicons from PSL extensions should be explicitly mapped to XML. Foundational theories and defined lexicons from PSL extensions should probably be omitted from the mapping, unless they describe containing or ordering relationships that can be easily represented implicitly.

Appendix D: Basic PSL Syntax

This section contains the definition of the PSL language using an extended Backus-Naur form (BNF).

BNF Conventions

The following extended BNF conventions are used:

- A vertical bar “|” indicates an exclusive disjunction; thus, for example, if C1 and C2 are two syntactic categories “C1 | C2” indicates an occurrence of either an instance of C1 or C2, but not both. The absence of such a bar between two constructs indicates a concatenation.
- An asterisk “*” immediately following a construct indicates that there can be any finite number (including 0) of instances of the construct.
- A plus sign “+” superscript immediately following a construct indicates that there can be one or more instances of the construct.
- Braces “{” and “}” are used to indicate grouping. Thus, “{C1 | C2}+” indicates one or more instances of either C1 or C2.
- A construct surrounded by brackets (e.g., “[C1 | C2]”) indicates that an instance of the indicated construct is optional.
- Terminals of a grammar appear in *courier* font. Nonterminals — representing categories of expressions — start with “<” and end with “>” are in Times Roman font. For example, “<b-var> ::= ?<b-indcon>” indicates that a variable must start with a question mark.
- Where necessary, the space character is represented by “<space>.”

Basic Tokens and Syntactic Categories

We first define a set of basic tokens and certain categories of expressions built up from them that will be used to define any first-order PSL language. They are defined in the following BNF.

```
<uc-letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y  
              | Z  
<lc-letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y  
              | z  
<letter> ::= <uc-letter> | <lc-letter>  
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
<oper> ::= - | ~ | # | $ | * | + | /  
<punct> ::= _ | - | ~ | ! | @ | # | $ | % | ^ | & | * | ( | ) | + | = | ' | : | ; | ' | < | > | , | . | ? | /  
            | | | [ | ] | { | } | <space>
```


An *expression* is any string of basic tokens. We define four basic categories of expression.

```

<b-con> ::=      {<lc-letter> | <digit>} {<letter> | <digit>}* {{ _ | - } {<letter> |
                  <digit>}+}*
<b-var> ::=      ?<b-con>[ ' ]
<b-func> ::=     {<oper> | <uc-letter>} {<letter> | <digit>}* {{ _ | - } {<letter> |
                  <digit>}+}*
<b-pred> ::=     {<uc-letter>} {<letter> | <digit>}* {{ _ | - } {<letter> | <digit>}+}*
<doc-string> ::= " {<letter> | <digit> | <punct> | \ " | \\ }* "

```

Thus, a <b-con> (i.e., an expression derived from the nonterminal <b-con>) is a string of alphanumeric characters, dashes, and underscores that begins with a lower case letter or digit and in which every dash and underscore is flanked on either side by a letter or digit. A <b-var> is the result of prefixing a <b-con> with a question mark and, optionally, appending a single quote (a “prime”). A <b-func> is just like a <b-con> except that it must begin with either an <oper>, a <punc>, or an upper case letter, and a <b-pred> is just like a <b-con> except that it must begin with an upper case letter. (Every <b-pred> is thus a <b-func>.) A <doc-string> is the result of quoting any string of tokens; double quotes and the backslash can be used as well as long as they are preceded by a backslash.

Lexicons

A first-order PSL language L is given in terms of a *lexicon* and a *grammar*. The lexicon provides the basic “words” of the language, and the grammar determines how the lexical elements may be used to build the complex, well-formed expressions of the language.

An PSL *lexicon* λ consists of the following:

- The expressions <space>, (,), not, and, or, =>, <=>, forall, and exists;
- A denumerable recursive set V_λ of <b-var>s (i.e., expressions derived from the nonterminal <b-var> in the above BNF), known as the *variables* of λ ;
- A recursive set C_λ of <b-con>s, known as the *constants* of λ which includes at least the strings inf-, inf+, max-, and max+.
- A recursive set F_λ of <b-func>s, known as the *function symbols* of λ , which includes at least the strings beginof and endof.
- A recursive set P_λ of <b-pred>s known as the *predicates* of λ , which includes at least the strings = activity, activity-occurrence, object, timepoint, is-occurring-at, occurrence-of, exists-at, before, and participates-in.

Grammars

Given an PSL lexicon λ , the *grammar based on* λ is given in the following BNF.

```

<con> ::=      a member of  $C_\lambda$ 
<var> ::=      a member of  $V_\lambda$ 
<func> ::=     a member of  $F_\lambda$ 

```

$\langle \text{pred} \rangle ::=$ a member of P_λ
 $\langle \text{term} \rangle ::=$ $\langle \text{atomterm} \rangle \mid \langle \text{compterm} \rangle$
 $\langle \text{atomterm} \rangle ::=$ $\langle \text{var} \rangle \mid \langle \text{con} \rangle$
 $\langle \text{compterm} \rangle ::=$ $(\langle \text{func} \rangle \langle \text{term} \rangle)$
 $\langle \text{sentence} \rangle ::=$ $\langle \text{atomsent} \rangle \mid \langle \text{boolsent} \rangle \mid \langle \text{quantsent} \rangle$
 $\langle \text{atomsent} \rangle ::=$ $(\langle \text{pred} \rangle \langle \text{term} \rangle) \mid (= \langle \text{term} \rangle \langle \text{term} \rangle)$
 $\langle \text{boolsent} \rangle ::=$ $(\text{not } \langle \text{sentence} \rangle) \mid (\text{and } \langle \text{sentence} \rangle \langle \text{sentence} \rangle) \mid (\text{or } \langle \text{sentence} \rangle \langle \text{sentence} \rangle) \mid (=> \langle \text{sentence} \rangle \langle \text{sentence} \rangle) \mid (<=> \langle \text{sentence} \rangle \langle \text{sentence} \rangle)$
 $\langle \text{quantsent} \rangle ::=$ $(\{\text{forall} \mid \text{exists}\} \{ \langle \text{var} \rangle \mid (\langle \text{var} \rangle +) \} \langle \text{sentence} \rangle)$

Languages

The *PSL language* L_λ , based on a lexicon λ , is the set of expressions that can be derived from the nonterminal $\langle \text{sentence} \rangle$ in the above grammar. The members of L_λ will also be called (appropriately enough) the *sentences* of L_λ . A *subsentence* of a given sentence ϕ of L_λ is a substring of ϕ (possibly identical with ϕ itself) that is also a sentence. An occurrence of a variable v in a sentence ϕ of L_λ is *bound* iff it is in a subsentence of ϕ that is of the form $(\text{exists } v \ \psi)$ or $(\text{forall } v \ \psi)$. Otherwise the occurrence is *free*. A sentence ϕ is *closed* iff no variable occurrence in ϕ is free. Let ψ be a sentence containing zero or more free occurrences of the pairwise distinct variables v_1, \dots, v_n , and let τ_1, \dots, τ_n be n pairwise distinct terms of L_λ . Then $\psi[v_1/\tau_1, \dots, v_n/\tau_n]$ is the result of replacing all free occurrences of each variable v_i in ψ with τ_i . A term τ is *free for* a variable v in ϕ iff no (free) occurrence of a variable in τ is bound in $\psi[v/\tau]$.

Defined Quantifiers

The following definitions are useful for expressing quantified propositions.

$(\text{forall } (v_1 \dots v_n : \psi) \ \theta) =_{df} (\text{forall } (v_1 \dots v_n) (=> \psi \ \theta))$
 $(\text{forall } (v_1 \dots v_n : \psi_1 \dots \psi_m) \ \theta) =_{df} (\text{forall } (v_1 \dots v_n) (=> (\text{and } \psi_1 \dots \psi_m) \ \theta))$
 $(\text{exists } (v_1 \dots v_n : \psi_1 \dots \psi_m) \ \theta) =_{df} (\text{exists } (v_1 \dots v_n) (\text{and } \psi_1 \dots \psi_m \ \theta))$
 $(\text{exists! } v \ \psi) =_{df} (\text{exists } (v : \psi) (\text{forall } v' (=> \psi[v/v'] (= v \ v'))))$,

where v' is free for v in ψ .

$(\text{exists! } (v_1 \dots v_n) \ \psi) =_{df} (\text{exists } (v_1 \dots v_n : \psi) (\text{forall } (\mu_1 \dots \mu_n) (=> \psi[v_1/\mu_1, \dots, v_n/\mu_n]))$

(and (= $v_1 \mu_1$) ... (= $v_n \mu_n$))))),

where each μ_i is free for v_i in ψ .

Essentially (exists! $v \psi$) says that exactly one thing satisfies condition ψ , and, more generally, (exists! ($v_1 \dots v_n$) ψ) says that exactly one sequence of n things satisfies ψ . For instance, at the time of this writing (1997), it is true that

```
(exists! (?x ?y)
  (and (US-president ?x)
        (US-vice-president ?y))),
```

since only the pair $\langle \text{Clinton}, \text{Gore} \rangle$ satisfy the condition (and (US-president $?x$) (US-vice-president $?y$)). Notice that we are talking about sequences here, not just sets. For instance, it is *not* true that

```
(exists! (?x ?y)
  (and (Texas-US-senator ?x)
        (Texas-US-senator ?y))),
```

since the condition (and (Texas-US-senator $?x$) (Texas-US-senator $?y$)) is satisfied by both sequences $\langle \text{Hutchison}, \text{Gramm} \rangle$ and $\langle \text{Gramm}, \text{Hutchison} \rangle$.

